SAMS

Освой самостоятельно

Perl

за 24 часа

Clinton Pierce

SAMS

Teach Yourself

Perl

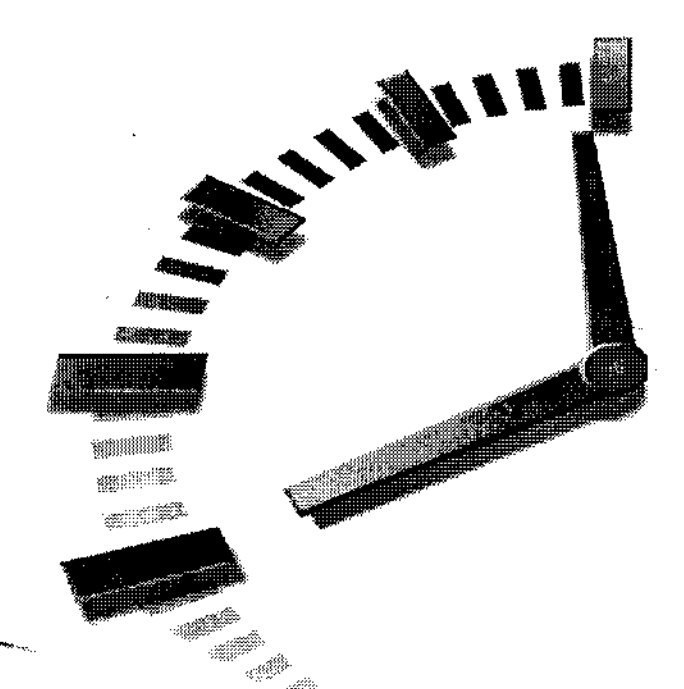
in 24 Hours

SAMS

A Division of Macmillan Computer

201 West 103rd St., Indianapolis, I

Клинтон Пирс



SAMS

Освой самостоятельно Perl за 24 часа



Издательский дом "Вильямс"
Москва • Санкт-Петербург • Киев
2001

ББК 32.973.26-018.2$_{я}$75
П19
УДК 681.3.07

Издательский дом "Вильямс"

По общим вопросам обращайтесь в Издательский дом "Вильямс"
по адресу: info@williamspublishing.com, http://www.williamspublishing.com

Пирс, Клинтон.

П19 Освой самостоятельно Perl за 24 часа. : Пер. с англ. : Уч. пос. — М. : Издательский дом "Вильяме", 2001. — 384 с.: ил. — Парал. тит. англ.

ISBN 5-8459-0097-2 (рус.)

Эта книга научит вас основам языка программирования Perl. Вы узнаете достаточно для того, чтобы самому сделать что-нибудь полезное. Автор ведет повествование в легком и доступном стиле, опуская в то же время редко встречающиеся нюансы программирования. Каждая новая методика продемонстрирована на множестве работающих примеров — от создания простейшей программы на Perl и до разработки сложных CGI-приложений.

Книга будет интересна широкому кругу читателей.

ББК 32.973.26-018.2$_{я}$75

ISBN 5-8459-0097-2 (рус.)
ISBN 0-672-31773-7 (англ.)

Оглавление

Содержание

Об авторе

Клинтон Пирс — специалист по разработке программного обеспечения, "вольный" программист и преподаватель. Уже пять лет он отвечает на вопросы по Perl, задаваемые в группах новостей Usenet, пишет программы курсов по Perl и обучает ему своих сотрудников, а также всех тех, кто хочет узнать об этом побольше. В те редкие часы, когда ему не приходится восстанавливать работоспособность системы UNIX, разрабатывать CGI-программы для компании Ford Motor, обучать пользователей работе с операционной системой UNIX в компании Decision Consultants или просто ради удовольствия сидя дома писать программы, Клинтон мечтает, чтобы его тайно похитили лесные нимфы и унесли подальше от цивилизации. Вы можете посетить его Web-сервер по адресу http://www.geeksalad.org.

Посвящение

Хейди и Кельвину за то, что они "не убили" меня этим летом и поддержали во время этой авантюры.

Без вашей поддержки я совсем утратил бы рассудок.

Благодарности

Чтобы видеть дальше, нужно стоять на плечах Гиганта.
— Исаак Ньютон

Поэтому в начале книги мы должны выразить признательность одному из Гигантов, сотворившему для нас Perl. Огромное спасибо тебе Лэрри[1].

Что же касается меня, то я не только стоял на плечах этого Гиганта, но и пользовался кладезем мудрости других людей. Чтобы написать хорошую книгу, содержащую минимум ошибок, я попросил совершенно незнакомых мне людей оценить мою рукопись и исправить ее, если в этом возникнет необходимость. Надо признаться, что я провел весьма поучительный эксперимент. В особенности мне хочется отметить тех, кто больше всего мне досаждал, выискивая в рукописи ошибки и предлагая готовые решения. Они заслуживают всяческой похвалы. Это — Абигайль, Грег Бейкен (Greg Bacon), Син Барк (Sean Bukre), Кен Фокс (Ken Fox), Кевин Мельтцер (Kevin Meltzer), Том Феникс (Tom Phoenix), Майкл Шверн (Michael Schwern), Том Гриделанд (Grydeland), Мэт Бьелански (Matt Bielanski), Марк Джейсон-Доминус (Mark Jason-Dominus), Джефф Пейниан (Jeff Pinyan), Гари Росс (Gary Ross), Эндрю Чен (Andrew Chen) и Джон Белл (John Bell).

Мне также хочется поблагодарить участников канала #perl за то, что они нашли время выслушать меня и высказали свое честное, а иногда и нелицеприятное мнение.

---

[1] *Речь идет о Лэрри Уолле (Larry Wall) — создателе языка Perl.* — Прим. ред.

В некотором смысле ответственность за книгу несут Билл Крауфорд (Bill Crawford) и Донна Хинкл (Donna Hinkle), из-за которых я вынужден был возиться со всеми этими упражнениями. Полагаю, я должен поблагодарить и их.

И, конечно же, я должен поблагодарить сотрудников издательства Macmillan, тех, кто терпеливо относился к начинающему автору и кто мужественно вынее все мои муки и·страдания. Без людей, типа Ренди Роджера (Randi Roger), Скотта Мейера (Scott Meyers), Чака Хатчинсона (Chuck Hutchinson) и многих других, кого я здесь не упомянул, мне бы никогда не удалось завершить начатое дело.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать электронное письмо или просто посетить наш Web-сервер, оставив свои замечания, — одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более подходящими для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш e-mail. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: info@williamspublishing.com

WWW: http://www.williamspublishing.com



В этой, как и в большинстве других книг по компьютерной тематике нашего издательства, есть листинги программ. Чтобы получить их, зарегистрируйтесь на нашем Web-сервере. Они будут вам присланы по электронной почте.

Введение

Любая достаточно развитая технология практически неотличима от магии.
— Артур С. Кларк

Так вот, зарубите себе на носу — *в программировании нет ничего магического и сверхъестественного.*

То, что кажется волшебством, как правило, лишь хитрый трюк, и программирование не является в этом смысле исключением. Некоторые аналитические способности, желание учиться и время для этого — вот все, что необходимо для обучения программированию на Perl. Поставьте перед собой цель. Для одного такой целью может быть разработка солидного Web-узла, другой желает сконвертировать уже имеющуюся у него программу на Perl, а кто-то просто интересуется этими вопросами из чистого любопытства — это не важно.

Что же может предложить вам эта книга, если вы уже выбрали цель и имеете все необходимое для изучения Perl?

Эта книга научит вас основам языка программирования Perl. Вы узнаете достаточно для того, чтобы самому сделать что-нибудь полезное. Мы не собираемся перегружать вас возможно интересными, но редко встречающимися нюансами программирования на Perl. Каждая новая концепция будет продемонстрирована на множестве работающих примеров. Вы сможете сами в этом убедиться.

А почему, собственно, нужно изучать Perl? Да потому, что этот язык используется практически в любой компании, которая хоть как-то связана с программированием. Perl нашел применение в финансовой области, производстве, генетике, военном деле, а также во всех остальных отраслях деятельности человека. И, конечно, где были бы Internet и World Wide Web без Perl? Судя по всему, Perl еще долго не сойдет с арены, поэтому то, что вы узнаете сейчас, сможет оказаться вам полезным в будущем.

С помощью Perl можно написать потрясающие программы, обходящиеся небольшим количеством кода. С его помощью можно свести воедино без особых трудозатрат разные языки программирования, приложения и программные технологии.

Как работать с этой книгой

Книга разбита на 24 темы, для изучения каждой из которых требуется приблизительно один час. Однако мы не хотим вас ограничивать этими рамками. Вы можете постигать урок за уроком хоть сутки напролет (если забудете про сон и аппетит) или проделывать это в собственном темпе, скажем за 10 минут,

В конце каждого часа вам будут предложены определенные задания. В уроках содержатся объяснения элементов и особенностей языка. В каждом уроке предусмотрена возможность удобного обучения путем выполнения предложенных упражнений.

Соглашения, принятые в этой книге

В книге *Освой самостоятельно Perl за 24 часа* приняты следующие соглашения.

- Каждый час предваряется его кратким обзором.
- Пошаговые инструкции выделены соответствующей пиктограммой.

- В конце каждого часа приведено резюме и список часто задаваемых вопросов и ответов на них. Будем надеяться, что среди них вы найдете вопросы, интересующие вас.

В книге также встречаются перечисленные ниже пиктограммы.



Заметки с комментариями и отступлениями от текущей темы

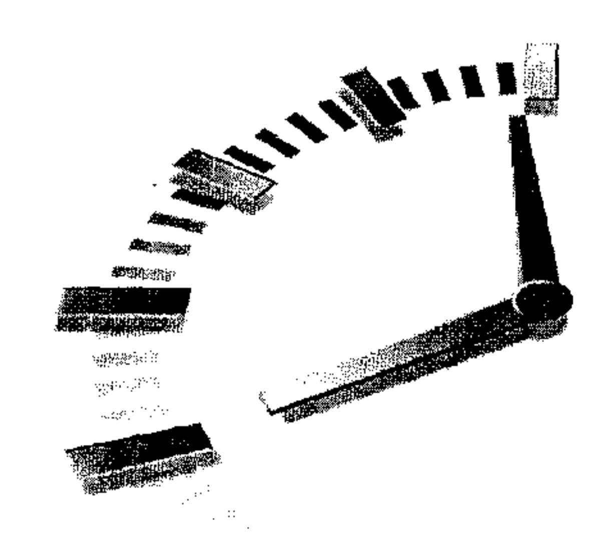


Рекомендации и советы помогут выполнить поставленное задание



Описываются типичные проблемы при работе с Perl и способы их устранения.

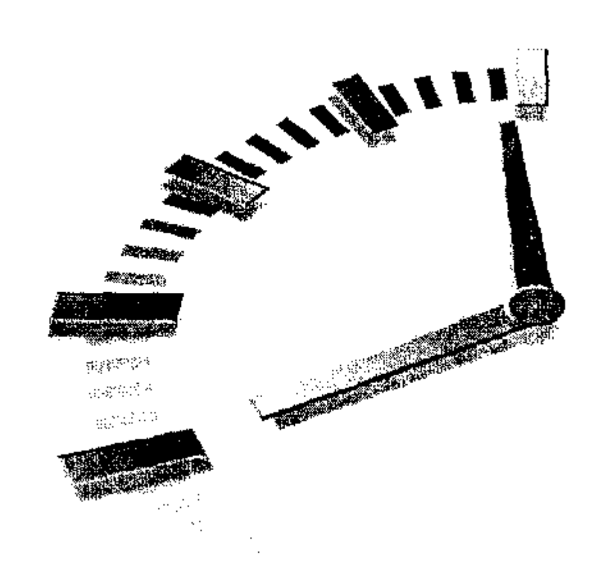
Новые термины выделяются *курсивом,* чтобы вы обратили на них внимание.



Часть I

Основы Perl

Темы занятий



1-й час

Начало работы с Perl

Perl — язык программирования общего назначения. Он может заменить любой язык программирования и применяться в любой области деятельности, которую только можно представить. Он используется для анализа рынка ценных бумаг, в производстве, конструировании, для поддержки пользователей, для контроля качества, тестирования программ на отсутствие проблемы 2000, системного программирования, проводки ведомостей, инвентаризации и, конечно же, в Web.

Perl получил такое широкое распространение потому, что он является *интегрирующим языком (glue language)*, который позволяет совместно использовать разнородные программные технологии. Например, на Perl не написан текстовый процессор лишь потому, что уже существует достаточное количество приличных текстовых процессоров и без Perl, а не потому, что это невозможно. На Perl вполне возможно написать приложение баз данных или электронных таблиц, операционную систему, полнофункциональный Web-сервер. Но весь вопрос в том, зачем это делать?

Сила Perl — в объединении вышеперечисленных элементов. Он может взять вашу базу данных, преобразовать ее для обработки в электронных таблицах и даже модифицировать необходимые данные. Perl способен преобразовать документы текстового процессора в формат HTML для их дальнейшего представления в Web.

Побочным следствием того, что Perl является интегрирующим языком, является его исключительная способность к адаптации. На данный момент он может работать, по крайней мере, с двумя десятками различных операционных систем. Стиль программирования Perl допускает большую гибкость, позволяя тем самым достичь цель разными способами. Ваши программы могут выглядеть совсем не так, как мои, но если они нормально работают, то все в порядке. Если нужно, Perl становится очень строгим языком или, наоборот, довольно снисходительным даже к начинающим программистам. Это ваш выбор.

Для начала отметим несколько важных моментов. Язык программирования называется *Perl.* Название программы (так называемого интерпретатора), которая запускает ваши программы, perl. Это различие не имеет сколько-нибудь важного значения до момента запуска ваших программ из командной строки. В этом случае вам следует воспользоваться командой perl. Кое-где, но только не в нашей книге, вы сможете встретить слово Perl, написанное как *PERL.* Название Perl на самом деле является аб-

бревиатурой от *Practical Extraction and Report Language,* что в переводе на русский язык означает *язык извлечения данных и подготовки отчетов.* В настоящее время уже не принято писать PERL, это слишком официально. Для друзей он просто Perl.



> Многие из элементов Perl заимствованы из других языков программирования. Эти заимствования привели к хождению следующей расшифровки названия Perl: *Pathologically Eclectic Rubbish Lister (патологически эклектичные мусорные грабли)*

Основные темы этого занятия.

- Установка Perl.
- Получение доступа к встроенной документации.
- Создание первого сценария.

Установка Perl

Прежде чем заняться изучением Perl, его нужно установить. Во время инсталляции Perl (несложной процедуры, едва ли допускающей возможность ошибок) запускается специальная программа тестирования, позволяющая убедиться в том, что инсталляция прошла успешно. Процедуры инсталляции отличаются в зависимости от используемой операционной системы. Поэтому сначала выясните, какая у вас операционная система.

А если Perl уже установлен?

Перед тем как перейти к решительным действиям по установке, следует проверить, не установлен ли уже Perl на вашем компьютере. Учтите, что интерпретатор Perl входит в стандартную поставку некоторых версий UNIX. В Windows NT программа Perl является частью Windows NT Resource Kit. Чтобы выяснить, имеется ли в вашей системе правильно установленная версия интерпретатора Perl, придется обратиться к командной строке.

В UNIX последовательность действий такова. Вы должны зарегистрироваться в системе и, если используется графический интерфейс, открыть окно терминала. В появившейся командной строке вы увидите символ приглашения, как показано ниже:

```
$
```

Иногда вместо символа $ можно увидеть символ % или `bash%`, что, по сути, одно и то же. После регистрации в системе оболочка выводит *приглашение командной строки* или *приглашение оболочки.* На первых нескольких занятиях вы будете взаимодействовать с Perl посредством командной строки.

Чтобы проверить, правильно ли установлен Perl в вашей системе, наберите следующую команду (символ $ набирать не надо):

```
$ perl -v
```

В ответ на эту команду система может выдать сообщение об ошибке {типа command not found) или же вывести номер версии интерпретатора Perl. В последнем случае считайте, что вам повезло — Perl у вас уже установлен и, скорее всего, переустанавливать его уже не потребуется.



Для успешной работы с Perl необходимо, чтобы номер версии его интерпретатора был не ниже 5 — 5.004, 5.005, 5.6 и т.д. Если у вас инсталлирован Perl версии *4.x,* вам придется его переустановить. В старой версии Perl было много ошибок и к тому же она больше не поддерживается. Кроме того, некоторые примеры, приведенные в этой книге, просто не будут работать. В момент подготовки русского издания настоящей книги текущей версией Perl была 5.6.

Если на вашем компьютере установлена система Windows, выяснить, правильно ли установлен у вас Perl, можно следующим образом. Запустите в окне сеанс MS-DOS, как показано на рис. 1.1. В командной строке наберите (приглашение C:\> набирать уже не нужно):

```
C:\>perl -v
```

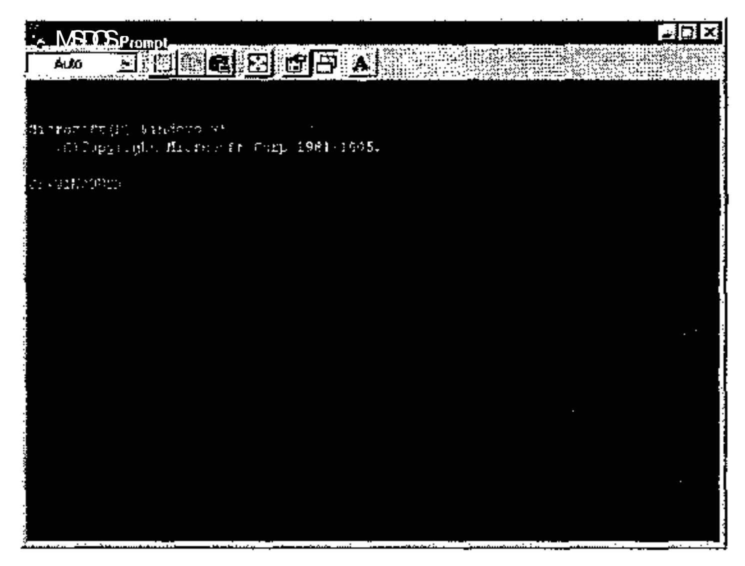


Рис. 1.1. Вы можете проверить номер версии Perl из командной строки MS-DOS

Если в ответ на эту команду будет выведен номер версии интерпретатора не ниже 5, то волноваться не о чем. Если же DOS выведет сообщение об ошибке Bad command or file name (Имя команды или файла указано неправильно), Perl нужно будет установить или переустановить (если используется версия ниже 5).

Проверить, установлен ли Perl на компьютере Macintosh, можно с помощью команды File Find (Command-f). При этом в качестве критерия поиска необходимо ввести строку MacPerl, как показано на рис. 1.2. Если приложение найдено, запустите его и выберите в меню Apple команду About MacPerl. Если у вас установлена версия не ниже 5.2.0 (Patchlevel 5.004), то все в порядке, в противном случае — придется установить новую версию MacPerl.

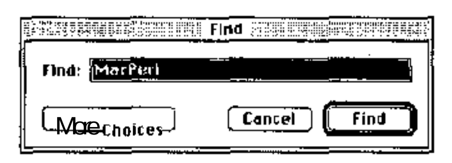


Рис. 1.2. Поиск Perl на Macintosh

Установка Perl в Windows 95/98/NT

Как часто бывает в жизни, для инсталляции Perl в Windows существует два способа — простой и сложный. Если вы знакомы с компилятором С и сопутствующими ему средствами разработки, такими как make-файлы, оболочки, — можете сами скомпилировать Perl из имеющихся исходных файлов. Они свободно распространяются, и вы можете их просмотреть и даже модифицировать на свой вкус. Если вы заинтересовались этим вопросом, обратитесь к материалу 16-го занятия, "Сообщество Perl", за необходимой информацией. Впрочем, стоит сразу отметить, что ручная инсталляция Perl в Windows не под силу рядовому пользователю.

Второй способ установки действительно не вызывает никаких сложностей. Компания ActiveState Tool распространяет Perl в виде приложения Windows, которое устанавливается подобно остальным приложениям Windows. На рис. 1.3 показано окно программы установки. Данный интерпретатор Perl распространяется на условиях общей лицензии сообщества ActiveState. Ознакомиться с лицензией можно по адресу `http://www.activestate.com`.

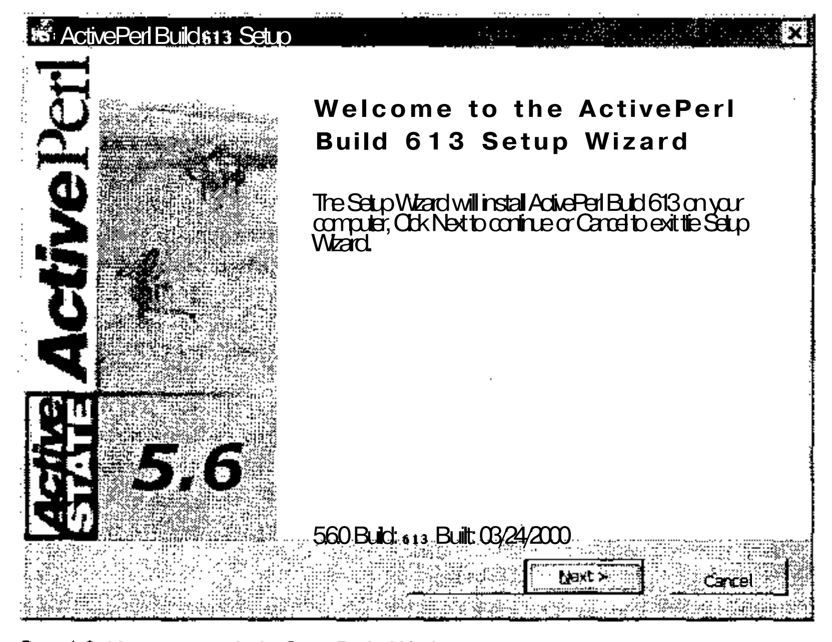


Рис. 1.3. Инсталляция ActiveState Perl в Windows



На прилагаемом к книге компакт-диске находится копия ActiveState Perl. Вы можете установить Perl с этого компакт-диска или загрузить последнюю версию программы с Web-сервера ActiveState.

Установка Perl в UNIX

Для установки Perl в UNIX у вас должны быть две вещи. Прежде всего, полный набор исходных файлов. Последнюю версию исходных файлов всегда можно загрузить из раздела Download Web-сервера http://www.perl.com. Там вы найдете сразу несколько

версий, но одна из них обязательно будет помечена как "Stable" или "Production". Вам также потребуется компилятор ANSI С. Не переживайте, если не знаете, что это значит. Программа конфигурации Perl сама проверит на наличие указанного компилятора, и в случае его отсутствия вы сможете установить скомпилированную версию, как описано в конце раздела.



Если в вашей UNIX имеется система для установки предварительно скомпилированных пакетов, вы можете установить бинарные коды пакета Perl без его компиляции. Кроме того, интерпретатор Perl обычно входит в поставку Linux, Solaris, AIX и многих других версий UNIX. Для получения информации о подобных пакетах обратитесь к документации.

Набор исходных текстов Perl содержится, как правило, в файле с именем stable.tar.gz. Перед установкой Perl его следует разархивировать. Для этого введите следующие команды:

```
$ gunzip stable.tar.gz
$ tar xf stable.tar
```

Для выполнения этих команд потребуется какое-то время. Если у вас нет программы разархивации gunzip, можете загрузить ее с сервера http://www.gnu.org. Программный пакет называется gzip. По окончании разархивации наберите в командной строке следующую команду:

```
$ sh Configure
```

При этом будет запущена программа Configure, которая задаст вам много вопросов. Если вы не знаете ответов на большинство из них — ничего страшного, просто нажмите клавишу <Enter>. Вам вполне подойдут параметры, принятые по умолчанию. Как правило, в большинстве UNIX-систем установка Perl не вызывает каких-либо проблем. После того как программа Configure завершит свою работу, введите команду

```
$ make
```

Она будет выполняться довольно долго — вы наверняка успеете выпить чашечку кофе. Если у вас маломощный компьютер, можете даже сделать перерыв на обед. После того как команда make завершит свое выполнение, введите еще две команды:

```
$ make test
t make install
```

Первая команда позволяет удостовериться, что компиляция Perl прошла нормально и он готов к установке. Чтобы запустить вторую команду, вам придется зарегистрироваться в системе как root, для которого приглашение обычно имеет вил #. Вторая выполнит установку Perl в системные каталоги.

Если команда make install корректно выполнила свою работу, вы можете окончательно убедиться в том, что установка Perl прошла успешно, повторно набрав в командной строке команду

```
$ perl -v
```

Если команда сработает — примите поздравления!



На прилагаемом компакт-диске содержится набор исходных файлов для установки Perl в UNIX. Вы можете воспользоваться ими или загрузить последнюю версию Perl с сервера http://www.perl.com.

Установка Perl на компьютерах Macintosh

Свежую версию Perl для Macintosh (MacPerl) можно загрузить с Web-сервера http://www.perl.com. Дистрибутивные файлы находятся в каталоге http://www.perl.cora/CPAN/ports/mac. Загрузите файл appl.bin из этого каталога, разархивируйте его программой StuffIt Expander, а затем запустите программу установки.

После этого вы должны активизировать программу для чтения документации Perl — Shuck, которая входит в комплект поставки MacPerl. Пользователи MacOS 8 могут сделать это с помощью Internet Control Panel, выбрав команду **Advanced⇨File** Mapping и ассоциировав расширение `.pod` с приложением Shuck. Это позволит получить удобный доступ к документации. Кроме того, полезно ассоциировать расширения `.ph`, `.pl`, `.plx`, `.pm`, `.cgi` и `.xs`, обычно назначаемые программой на Perl, с приложением MacPerl. Установите для них тип файла TEXT.

Пользователи MacOS 7 для выполнения описанных выше действий должны воспользоваться утилитой InternetConfig. В InternetConfig выберите команду Helpers и свяжите расширение `.pod` с приложением справки Shuck, а также свяжите все расширения Perl с приложением MacPerl.



На прилагаемом компакт-диске находится копия дистрибутивного пакета MacPerl. Установите MacPerl с него или загрузите последнюю версию с узла `http://www.perl.com/CPAN/ports/mac`.

Документация

А теперь обратите внимание: в комплект поставки Perl входит *полная* версия текущей документации по языку и интерпретатору Perl.

Да, да! Вы не ошиблись. Устанавливается именно *полная* версия документации. Причем бесплатно! В поставку Perl версии 5.6 входит более 1700 страниц документации! Документация содержит справочный материал, учебники, список часто задаваемых вопросов и ответов на них (так называемый FAQ), историю развития и даже примечания, касающиеся внутреннего устройства Perl.

Получить доступ к документации можно несколькими способами. В Windows и UNIX вместе с Perl инсталлируется утилита `perldoc`. Ее можно использовать как инструмент поиска нужного раздела документации и получения форматированного вывода. Для запуска perldoc необходимо перейти в окно командной строки. В следующем примере мы покажем действие утилиты в системе UNIX, но в DOS все происходит точно так же.

```
$ perldoc perl
PERL(1)        User Contributed Perl Documentation        PERL(1)


NAME
       perl - Practical Extraction and Report Language

SYNOPSIS
       perl [ -sTuU ]      [ -hv ] [ -V[:configvarl ]
            [ -cw ] [-d[:debugger] ] [-D[number/list] ]
            {-pna ] [-Fpattern ] [ -l[octal] J [-0[octal] ]
            [-Idir ] [ -m[-]module ] [ -M[-]'module...' ]
            [ -P ]     [ -S ]      [ -x[dir] )
```

```
[ -i[extension] ]
[ -e 'command' ] [ — ] [ programfile ] [ argument ]...
```

```
For ease of access, the Perl manual has been split up into
a number of sections:
```

Разделы руководства состоят из отдельных частей, которым присвоены имена, например: perlfunc (функции Perl), perlop (операторы Perl) или perlfaq (список часто задаваемых вопросов по Perl). Для доступа к странице руководства perlfunc введите команду perldoc perlfunc. Названия всех разделов руководства перечислены на странице руководства perldoc perl.

Чтобы найти в руководстве функцию по имени, нужно запустить утилиту perldoc с ключом -tf. В следующем примере показано, как найти в руководстве описание функции print:

```
$ perldoc -tf print
```

В FAQ собраны часто задаваемые вопросы, касающиеся Perl. Люди, изучающие Perl, задают одни и те же вопросы. Поэтому, чтобы сохранить время и избавить их от множества проблем, все эти вопросы собраны в файлы FAQ. Чтобы найти нужную тему в FAQ, воспользуйтесь ключом -q, после него укажите слово, которое может находиться в заголовке FAQ. К примеру, если вы хотите узнать о поддержке Perl, введите следующую команду:

```
$ perldoc -q support
```

В результате выполнения этой команды отобразится страница FAQ Who supports Perl? Who develops it? Why is it free?

Другие способы доступа к документации

Во время установки Perl в UNIX-системе вам предоставляется возможность установить документацию в традиционном man-формате. При этом стандартная документация по Perl будет конвертирована в формат man и помещена в соответствующий каталог. В результате для доступа к документации по Perl можно будет использовать и программу perldoc, и программу man, как это обычно принято в UNIX:

```
$ man perl
```

При установке в Windows пакета ActiveState Perl страницы справочного руководства конвертируются в формат HTML, в результате их можно просмотреть в любом Web-броузере, который поддерживает фреймы. При стандартной установке файлы документации находятся в каталоге `C:\Perl\html`. Если вы установили Perl в другой каталог, ищите файлы документации в подкаталоге html.

В пакет MacPerl входит утилита Shuck, которая находится в папке MacPerl. Используйте ее для поиска и чтения разделов документации, как показано на рис. 1.4.

А если документация отсутствует?

Существуют лишь две причины, по которым вы не сможете найти документацию. Во-первых, нужно знать, где искать. Утилита perldoc может быть расположена в каталоге, отсутствующем в списке путей оболочки. В этом случае найдите программу perldoc и добавьте путь к ее каталогу в переменную окружения оболочки PATH. Во-вторых, документация могла быть нечаянно или по злому умыслу удалена. Любая установка Perl обязательно содержит документацию, более того, Perl никогда не инстал-

лируется без документации. Ее отсутствие свидетельствует о том, что пакет Perl или с самого начала неправильно установлен, или впоследствии был поврежден. В этом случае, вероятно, вам (или системному администратору) придется переустановить Perl. Документация является важной составляющей среды разработки Perl и без нее некоторые части Perl просто не будут функционировать.

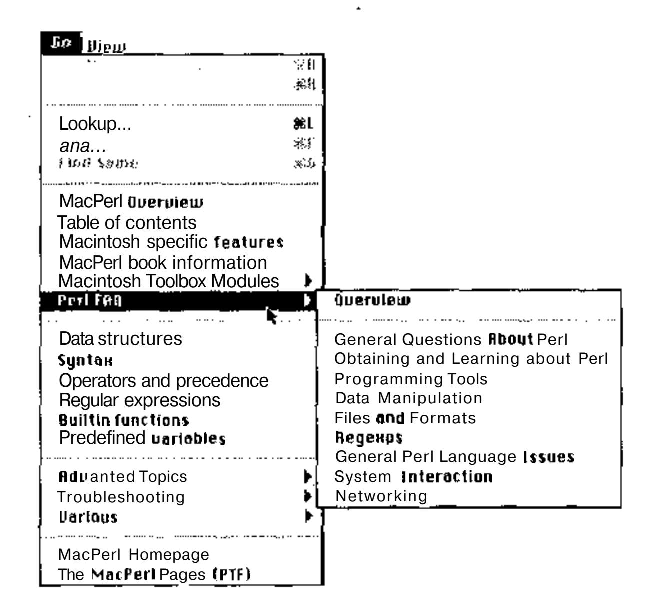


Рис. 1.4. Утилита Shuck используется для просмотра документации по MacPerl

Если еше что-либо произошло и вы не можете получить доступ к локальной копии документации, вам придется снова обратиться к Web-серверу. На главной странице http://www.perl.com есть ссылка на страницу документации. Конечно, лучше всегда иметь под рукой документацию, относящуюся к вашей конкретной версии Perl, но в крайнем случае подойдет и эта.

Ваша первая программа

Чтобы написать программу на языке Perl, необходимо иметь *текстовый редактор,* который позволяет набрать неформатированный текст и сохранить его в файле. В качестве примера простых текстовых редакторов можно привести программу Notepad (Блокнот) в Microsoft Windows и EDIT.EXE в DOS. В UNIX имеются текстовые редакторы vi, emacs и pico. По крайней мере, один из них должен быть на вашем компьютере. На Mac приложение MacPerl содержит встроенный текстовый редактор, поэтому для создания новой программы достаточно выбрать команду File⇨New.

Вам *не* следует для набора программ на Perl пользоваться текстовым процессором. Текстовые процессоры (такие как Microsoft Word, Wordpad, WordPerfect и др.) сохраняют документы вместе с информацией об их форматировании, даже если вы не используете специальных атрибутов форматирования. Интерпретатор Perl не понимает эти коды и поэтому программы, набранные в текстовом процессоре, не будут работать. Если все же вы решите воспользоваться текстовым процессором, не забудьте сохранить программы в виде обычного текстового файла.

Наберите вашу первую программу

Откройте текстовый редактор и наберите следующую программу:

```
#!/usr/bin/perl

print "Hello, World!\n";
```

Строка с #! должна быть первой строкой файла.

После того как вы наберете эту программу в текстовом редакторе, сохраните ее в файле под названием hello. Расширение не обязательно, но если хотите, можете его указать. Некоторые приложения Windows и Macintosh используют расширения для определения типа файлов. Если по какой-то причине вы решили дать расширение, то лучше воспользуйтесь стандартными — .pl или .plx. Для определенности назовите набранный файл hello.pl.

Запуск программы

Для запуска программы необходимо перейти в режим командной строки. В UNIX зарегистрируйтесь в системе и откройте окно терминала. В Microsoft Windows откройте сеанс MS-DOS. Перейдите в каталог, где находится программа hello, используя команду оболочки cd.

Наберите в командной строке следующее. (Здесь показана командная строка DOS, приглашение командной строки UNIX несколько отличается.)

```
C:\PROGRAMS> perl hello
```

Должно появиться сообщение:

```
Hello, World!
```

Если у вас получилось, примите наши поздравления! Запомните способ запуска этой программы, потому что подобным образом мы будем запускать программы на протяжении всей книги. (Существуют и другие способы, мы вам расскажем и о них.)

Если что-то не получилось, проверьте следующее.

- Если появилось сообщение об ошибке Bad command or file name или perl: command not found, посмотрите, в каком каталоге находится программа perl, и добавьте путь к ней в переменную оболочки PATH.

- Сообщение об ошибке Can't open perl script hello: A file or directory does not exist говорит о том, что либо файл hello находится в другом каталоге, либо вы сохранили его под другим именем.

- Ошибка syntax error означает, что интерпретатор Perl нормально запустился, но программа не может определить, что же находится в файле hello. Например, вы могли ошибиться при наборе, или тестовый процессор добавил в сохраненный файл коды форматирования, Для просмотра содержимого файла в UNIX воспользуйтесь командой cat, а в DOS — type. Тщательно проверьте содержимое файла, обращая внимание на кавычки и знаки пунктуации.

Если вы используете пакет MacPerl, просто выберите команду Run "hello" в меню Script, и ваша программа запустится. Если для набора программы вы не пользовались встроенным тестовым редактором, откройте программы с помощью команды Open меню File, а затем выполните команду Run.

Заработало! Что же произошло?

Команда perl hello запустила на вашем компьютере программу perl. Эта программа называется *интерпретатором perl.* Интерпретатор — "ум, честь и совесть" Perl. Его задача состоит в том, чтобы открыть указанный в командной строке файл (в данном случае hello), найти в нем программу и выполнить ее.

Под фразой "выполнить ее" я подразумевал следующее. Прежде всего интерпретатор должен проверить корректность синтаксиса выражений, функций и операторов, составляющих программу на Perl, а затем последовательно все их выполнить.

Интерпретатор perl читает программу с диска и выполняет ее до тех пор, пока она не закончится. После выполнения программы интерпретатор возвращает управление операционной системе.

Теперь посмотрим, как же выполнялась программа hello.

Проследим за Perl

Первая строка программы hello

```
#!/usr/bin/perl
```

В Perl все, что идет в строке после символа #, считается комментарием. Комментарии Perl попросту игнорирует. Однако символы #! означают нечто другое. После них должен быть указан путь к интерпретатору perl — /usr/bin/perl. В UNIX принято, что программа, содержащая в первой строке символы #!, после которых указан путь к интерпретатору, может запускаться по имени. Обратитесь к разделу "Вопросы и ответы" в конце этого занятия за информацией о запуске программ. Некоторые Web-серверы, например Apache, также используют строку с #! для запуска программ без непосредственного вызова интерпретатора perl.

Для нас сейчас это не важно, мы считаем первую строку комментарием.

Следующая строка программы:

```
print "Hello, World!\n";
```

В этой строке находится оператор Perl, обозначающий действия, которые должен выполнить Perl. Строка содержит вызов функции print, которой передается строка Hello, World! и символ новой строки. Функция должна отобразить эту строку на экране. В конце оператора следует точка с запятой.

В Perl символ ; называется *оператором-разделителем*. Он разделяет операторы в программах Perl и указывает, где конец одного оператора и начало следующего.

В нашем случае функция print отображает фразу Hello, World!. Символы \n в конце строки говорят Perl о том, что нужно вывести пустую строку после напечатанной фразы. Кавычки, в которые заключена фраза и \n, сообщают Perl о том, что это строка, а не другая функция. На следующем занятии мы посвятим строкам достаточно много времени.

Это вы должны знать

Программы на Perl можно записывать в свободной форме. Синтаксис языка не требует каких-либо особых форм записи или форматирования текста. Вы можете помещать между операторами пробелы, табуляторы и даже символы перехода на новую строку. В Perl все они называются *пробелами* и никак не влияют на работу интерпретатора.

Конечно, есть определенные места в программе, где нельзя помешать такие символы. Например, нельзя вставлять пробелы в имя функции: pr int — неправильное имя функции. Нельзя также вставлять их в числа, скажем, 25 61 — не то же самое, что 2561. Пробелы внутри строки, как, например, в строке "Hello, World!", всегда отображаются. За исключением приведенных выше случаев, во все остальные места программы можно безболезненно помешать символы пробела. Например, вы можете переписать программу из рассматриваемого нами примера следующим образом:

```
#!/usr/bin/perl

print
    "Hello, World!\n"
    }
```

Она будет функционировать точно так же, как и оригинал. Эта некоторая вольность Perl позволяет выбирать из множества стилей написания программ тот, который больше всего вам придется по душе. Вы можете совершенно свободно выразить себя в этом. Не забывайте только и о других пользователях, которые, возможно, когда-нибудь обратятся к вашим программам. Постарайтесь сделать ваш код понятным и для них.

Стиль, используемый в наших примерах, достаточно консервативен. Иногда операторы разделяются несколькими пустыми строками, чтобы разделить логические блоки в достаточно длинной программе на Perl. Кстати, в документации по Perl приведено описание рекомендуемого стиля оформления программ. Документ этот называется peristyle.



Стили оформления программ в Peri могут быть весьма вычурными. Программы могут быть написаны в столбик, как стихотворение, или в одну строку. Некоторые программисты умудряются оформлять программы в виде рисунка, и, что самое интересное, программы при этом не только не перестают работать, но еще и выполняют полезную работу! Каждый год *The Perl Journal* (http://www.tpj.com) проводит конкурс на самую запутанную Perl-программу — Obfuscated Perl Contest. Лучше бы вам этого не видеть.

Резюме

На этом занятии вы немножко окунулись в мир Perl и увидели, как работает интерпретатор. По мере чтения книги вы будете узнавать все больше и больше подробностей. Вы узнали, как инсталлировать Perl, проверить правильность его работы и убедиться, что все установлено должным образом. Вы набрали и, надеемся, запустили свою первую программу на Perl и разобрались, как она работает.

Вопросы и ответы

Как называются файлы, которые запускает Perl, — сценарии или программы?

Название не имеет особого значения. Обычно *программы* компилируются в машинный код, который сохраняется в специальном файле, допускающем запуск средствами операционной системы. Сценарии же загружаются во внешнюю программу, которая выполняет их каждый раз, когда это требуется. Создатель Perl Лэрри Уолл как-то сказал: "Сценарий — это то, что дают актеру, а программу раздают зрителям". Как хотите — так и понимайте. В этой книге я буду использовать термин *программа на Perl,* и если вы будете стараться, то, прочитав данную книгу, сможете гордо называть себя программистом на Perl.

Необходимо ли все время вручную набирать листинги этой книги? Некоторые из них довольно длинные.

Все тексты программ и примеров из этой книги со всеми требуемыми файлами данных содержатся на прилагаемом компакт-диске.

В разделе "Запуск программы" вы упоминали о более простом способе запуска программ в UNIX. Что это за способ?

Сначала убедитесь, что в первой строке файла после символов `#!` указан правильный путь к интерпретатору Perl, как правило, `/usr/bin/perl` или, на некоторых машинах, `/usr/local/bin/perl`. Далее вы должны сделать ваш файл выполняемым с помощью команды `chmod`. Для программы hello команда оболочки UNIX будет выглядеть как `chmod +x hello`. После этого вы сможете запускать программу на Perl командой hello или `./hello`.



Не называйте в UNIX программу именем `test`, поскольку в оболочке UNIX уже имеется команда с таким именем. В результате вы не сможете запустить свою программу. Обратитесь к документации по оболочкам, чтобы узнать, каких еще имен следует избегать.

Если вы хотите воспользоваться готовыми файлами с листингами программ, которые приведены на компакт-диске, не забудьте изменить первую строку программы и указать после символов `!#` путь к интерпретатору Perl вашей системы. В противном случае вам придется запускать программы из командной строки с помощью команды `perl` *имя_программы*, как было описано выше, в разделе "Запуск программы".

Семинар

Контрольные вопросы

1. Perl — название языка, а perl — это:

   а) также название языка;

   б) интерпретатор;

   в) команда DOS.

2. Откуда всегда можно загрузить копию документации по Perl?

   а) `http://www.microsoft.com`

   б) `http://www.perl.com`

   в) `http://www.perl.net`

3. На какой странице руководства можно найти описание синтаксиса Perl?

   а) `persyn`

   б) `perlop`

   в) `perlfaq`

Ответы

1. Правильный ответ б), но, поскольку, после установки perl становится командой оболочки DOS, вариант в) также допустим.
2. Правильный ответ — вариант б). А также в вашей системе.
3. Правильным будет вариант а), хотя проверить можно, лишь запустив команду perldoc perl, не правда ли?

Упражнения

- Попробуйте бегло просмотреть список часто задаваемых вопросов по Perl. Даже если сначала вам многое покажется непонятным, вы получите представление о том, какого рода информация содержится в FAQ.
- Если вы предпочитаете использовать Web-броузер, "сходите" на узел http://www.perl.com и прочитайте документацию там, только не переусердствуйте.

2-й час

Строительные блоки Perl: числа и строки

Любой язык, будь то компьютерный или "человеческий", предполагает наличие предмета разговора. Perl в основном имеет дело с числами и строками, их объединяющее название — *скаляры.*

Скаляры — базовый тип Perl. На каждом из занятий этой книги с ними будут производиться операции суммирования, вычитания, поиска, проверки, сбора, очистки, разделения на части, упаковки, сортировки, сохранения, загрузки, печати и удаления.

В Perl каждый скаляр (или переменная скалярного типа) может содержать отдельное слово, запись, документ, строку текста или символ. Скаляры Perl могут содержать также *литеральные данные,* т.е. данные, значение которых не изменяется во время выполнения программы. В некоторых языках программирования для такого рода данных используются термины *константа* или *литерал.* Они часто используются для хранения значений, не подверженных изменению в силу своей природы, например числа л, или ускорения земного притяжения g, или же имени 15-го президента США. Все подобные числа в программах Perl встречаются в виде скалярных литералов.

Другим типом скаляров Perl являются так называемые *скалярные переменные.* Переменные содержат данные, которыми можно свободно манипулировать. Вы можете изменять содержимое переменных, поскольку они являются всего лишь удобным инструментом доступа к данным, которые в них хранятся. Обычно переменным назначают удобные и легко запоминающиеся имена, которые связаны с хранящимися в них данными.

На этом занятии вы также познакомитесь с операторами Perl. *Операторы* — это своего рода глаголы языка Perl. Они оперируют существительными Perl таким образом, чтобы программа выполняла полезные действия.

Основные темы этого занятия.

- Числовые и строковые константы (литералы).
- Скалярные переменные.
- Операторы.

Литералы

Perl содержит два различных типа скалярных констант, называемых *литералами:* числовые и строковые литералы.

Числовые литералы

Числовые литералы — это обычные числа. Perl "понимает" числа, представленные в различных форматах. Все примеры, приведенные в табл. 2.1, являются допустимыми числовыми литералами Perl.

Таблица 2.1. Примеры числовых литералов

Число	Тип
6	Целое число
12.5	Вещественное число
15.	Еще одно вещественное число
.7320508	И еще одно вещественное число
1e10	Вещественное число, представленное в экспоненциальном формате (число с плавающей точкой)
6.67E-33	Еще одно число с плавающей точкой (допустимы как символы e, так и E)
4_294_296	Большое целое число. В качестве разделителя тысяч вместо запятых используется символ подчеркивания

Я думаю, здесь нет смысла описывать, что такое числа, поскольку это вам должно быть известно из курса средней школы. Остановимся только на нюансах. Целые числа представляются набором нескольких цифр. В вещественных числах используется десятичная точка. Числа с плавающей точкой содержат мантиссу, букву e и порядок числа. В больших числах для удобства чтения можно использовать символ подчеркивания, разделяющий тысячи. Перед тем как использовать такие числа, Perl удаляет символы подчеркивания.



Не ставьте перед числами нули, как, например, в 010. В Perl так обозначаются числа в восьмеричной системе счисления. В Perl также можно использовать числа в шестнадцатеричной и двоичной системах счисления. Подробней обо всем этом можно узнать, обратившись к разделу документации `perldata`.

Строковые литералы

Строковые литералы в Perl — это обычные строки символов. Они могут содержать любое количество данных. Размеры строк ограничены лишь размерами виртуальной памяти вашего компьютера. Строки могут содержать данные различных типов — текст ASCII со стандартным набором символов, текст ASCII с полным набором символов и даже двоичные данные. Строки могут быть также пустыми.

В Perl, за небольшими исключениями, вы обязаны заключать строковые литералы в кавычки. Этот процесс называется *квотингом (quoting)* строки. Для этого можно использовать или одинарные (' '), или двойные кавычки (" ")- Ниже приведено несколько примеров строковых литералов:

"Привет!"
'Дело было в прошлом веке'
"Одна рыба,\nВторая рыба,\nКрасная рыба,\nКуча раков\n"
""
"Мой дорогой Мишель, постарайся все сделать быстро.\n"

Если внутри двойных кавычек нужно поместить символ кавычки, перед ним необходимо поставить символ обратной косой черты. Этот символ, используемый внутри строкового литерала, говорит Perl, что идущий за ним символ не должен восприниматься как управляющий. Посмотрите на следующую строку, которую Perl поймет неправильно:

"И тогда я сказал ему: "Иди и принеси мне это.""

В ней кавычка перед словом Иди соответствует первой кавычке строки, поэтому фраза Иди и принеси мне это. остается за кавычками — так в Perl писать нельзя. Для предупреждения подобной ситуации поставьте перед кавычками, которые Perl должен проигнорировать, символ обратной косой черты, как показано ниже:

"И тогда я сказал ему: \"Иди и принеси мне это.\""

Символы обратной косой черты указывают Perl, что следующий за ним символ кавычки не закрывает строку, а является обычным текстовым символом. Это же правило может быть применено и к одинарным кавычкам, например:

'Перед одинарной кавычкой V поставьте обратную косую черту.'

Основное отличие между одинарными и двойными кавычками заключается в том, что строка в одинарных кавычках является литералом в чистом виде. Поэтому ее содержимое никак не интерпретируется Perl. В строках, заключенных в двойные кавычки, могут находиться имена переменных и *последовательности управляющих символов.* Эти последовательности имеют специальное обозначение, что позволяет включать в строки такие символы, которые сложно или невозможной набрать с клавиатуры или ввести каким-либо другим способом. В табл. 2.2 приведен краткий список управляющих последовательностей Perl.

Таблица 2.2. Примеры управляющих последовательностей

Обозначение	Описание
\п	Новая строка
\r	Возврат каретки
\t	Табулятор
\b	Символ забоя
\u	Поднятие регистра следующего символа
\1	Понижение регистра следующего символа
\\	Обратная косая черта в литералах
V	Одинарная кавычка внутри строки в одинарных кавычках
V	Двойная кавычка внутри строки в двойных кавычках



Полный список управляющих последовательностей можно найти в справочной документации. На предыдущем занятии я уже рассказывал, как обратиться к документации по Perl с помощью встроенной утилиты perldoc. Управляющие последовательности описаны в руководстве под рубрикой perlop в разделе Quote and Quote-like Operators.

При наборе строки с большим количеством кавычек очень легко допустить ошибку, поскольку перед каждой внутренней кавычкой нужно обязательно поставить обратную косую черту, как показано ниже:

"И тогда я сказал: \"Иди вперед\", а он ответил: \"Слушаюсь!\"."

Поэтому для облегчения процесса квотинга в Perl предусмотрены специальные операторы qq и q. Оператор qq заменяет двойные кавычки и ведет себя почти во всех случаях точно так же, как пара двойных кавычек:

qq{И тогда я сказал: "Иди вперед", а он ответил: "Слушаюсь!".)

Одинарные кавычки могут быть заменены оператором q:

q('Литералы' нужно заключить в одинарные кавычки)

Для обозначения начала и конца строк в операторах qq и q можно использовать любые символы, кроме алфавитно-цифровых. Эти символы называются *ограничителями (delimiters)*. В предыдущих примерах я воспользовался скобками, но, как я уже говорил, допустимы любые символы, кроме алфавитно-цифровых:

q/'Литералы' нужно заключить в одинарные кавычки/
q,'Литералы' нужно заключить в одинарные кавычки,

Символы, которые вы хотите использовать в качестве ограничителей, следует указать сразу после операторов qq или q. Вы можете использовать парные символы — (),<>, {}, (] со строками, содержащими эти символы. Причем, если они идут парами внутри операторов qq и q, обратная косая черта не используется:

q(Джо (отец Тона) рубил все деревья подряд (кроне самых больших).);

Использование вложенных скобок или других символов может снизить читабельность программы. Поэтому обычно используют ограничители, символы которых не встречаются в строке:

q[Джо (отец Тока) рубил все деревья подряд (кроне самых больших).];

Скалярные переменные

ДЛЯ хранения скалярных данных в Perl используются скалярные переменные. Скалярная переменная обозначается символом доллара и следующим за ним именем переменной. Вот несколько примеров скалярных переменных:

```
$a
$total
$Date
$serial_number
$cat450
```

Символ доллара, называемый *идентификатором типа*, указывает Perl, что переменная содержит скалярные данные. Кроме переменных скалярного типа, в Perl существуют также переменные других типов (хэши и массивы), для обозначения которых предусмотрены специальные символы. Кроме того, в некоторых случаях в именах переменных идентификаторы могут вообще не использоваться (например, для обозначения дескриптора файла). Имена переменных в Perl, независимо от их типа, т.е. имена хэшей, массивов, дескрипторов файлов и скаляров, должны следовать определенным правилам.

- Имена переменных должны состоять из идентификатора типа и идущих за ним символов латинского алфавита (a–z и A–Z), цифровых символов или символа подчеркивания. Но первый символ переменной не может быть цифрой.
- Имена переменных чувствительны к регистру. Это означает, что для имен переменных немаловажно, в какой форме идет буквенный символ: строчной или прописной. Следующие имена представляют различные скалярные переменные:

```
$value

$VALUE

$Value

$valuE
```

Кроме того, в Perl зарезервированы имена некоторых односимвольных переменных. Такие переменные, как $_, $", $/, $2 и $$, называются специальными и их не следует использовать как обычные переменные в Perl-программах. Назначение специальных переменных будет описано ниже.

В отличие от некоторых других языков программирования, в Perl переменные перед использованием не обязательно должны быть описаны и проинициализированы. Для создания скалярной переменной просто используйте ее. Для неинициализированных переменных Perl использует значение, принятое по умолчанию. Если переменная используется как число (например, в математическом выражении), ее значение по умолчанию — 0 (нуль), если переменная используется как строка (т.е. почти во всех остальных случаях), используется "" или пустая строка.



Использование неинициализированных переменных считается образцом плохого стиля программирования. Если Perl запущен из командной строки с ключом -w или же этот ключ указан в первой строке программы (с символами K!) после имени интерпретатора, подобные ситуации отслеживаются и выдается соответствующее предупреждение. Если вы используете неинициализированную переменную, выдается ошибка Use of uninitialized value.

Специальная переменная $_

В Perl предусмотрена специальная переменная, значение которой используется во многих выражениях, если явно не указана другая переменная. Речь идет о переменной $_. Например, если вызвать функцию print без параметров, будет распечатано текущее значение переменной $_:

```
$_="Dark Side of the Moon";
print;   #Печатает значение переменной $_, "Dark Side ..."
```

Подобное неявное использование переменной $_ может создать определенную неразбериху, учитывая, что этой переменной значение может присваиваться задолго до ее применения.

Однако подобная методика удобна в случае применения регулярных выражений, о чем речь пойдет на 6-м занятии, "Поиск по шаблону". В этой книге я старался как можно реже использовать переменную $_, чтобы программы было легче читать.

Выражения и операторы

После того как вы познакомились со скалярными типами данных, можно попробовать сделать что-то полезное на Perl. Программа Perl — набор выражений и операторов, выполняемых последовательно от начала и до конца, если, конечно, вы не изменяете ход программы с помощью специальных операторов, описанных на 3-м занятии, "Управление процессом выполнения программы". Пример готовой программы на Perl приведен в листинге 2.1.

Листинг 2.1. Пример программы на Perl

```
1: #!/usr/bin/perl -w
2:
3:  $radius=50;
4:
5: $area=3.14159*($radius ** 2);
6:   print $area;
```



Проведем анализ программы.

- *Строка 1.* Как вы помните, в этой строке указывается путь к интерпретатору Perl. Ключ `-w` говорит о том, что нужно выводить все предупреждения.
- *Строка 3.* В этой строке выполняется оператор присваивания. Скалярной переменной `$radius` присваивается число 50.
- *Строка 5.* В этой строке также выполняется оператор присваивания. В правой части оператора присваивания находится *выражение.* Выражение содержит скалярную переменную `$radius`, операторы * и ** (ниже описано их действие) и числовой скаляр 2. Переменной `$area` присваивается вычисленное значение выражения.
- *Строка 6.* В этой строке распечатывается результат вычислений, находящийся в переменной `$area`.

Выражение — это набор операторов, имеющих значение. Например, 2 — допустимое выражение, как и 54*$r, "Java", sin($pi*8) и $t=6. Значения выражений вычисляются в ходе выполнения программы. Вначале программа вычисляет значения функций, операторов и скалярных констант, составляющих выражение, а затем — результирующее значение. Выражения можно использовать в операторах присваивания, в других выражениях или в операторах Perl.

Основные операторы

Как было показано в листинге 2.1, для присваивания используется оператор присваивания =. Этот оператор помещает значение выражения, находящегося в правой части, в переменную, указанную в левой части. Например:

```
$title="Унесенные ветром";
$pi=3.14159;
```

Оператор, находящийся в правой части, должен иметь определенное значение, которое можно присвоить переменной, т.е. правая часть оператора присваивания должна являться выражением. Сама по себе операция присваивания также является выражением,

значение которого указывается в правой части оператора присваивания. Это означает, что в приведенной ниже строке кода переменным $a, $Ь и $c присваивается значение 42:

```
$a=$Ь=$c=42;
```

В этой строке переменной $c присваивается значение 42, затем $b присваивается значение выражения $c=42 (равное 42). Переменной $a присваивается значение выражения $b=42. Переменная, которой присвоено значение, может использоваться в правой части оператора присваивания, как показано ниже:

```
$a=89*$a;
$count=$count+1;
```

Правая часть выражения оператора присваивания вычисляется с использованием старого значения переменной $a или $count. Оператор присваивания во второй строке имеет специальное название в Perl — *инкремент.* Мы еще остановимся на таких операторах подробнее.

Числовые операторы

У Perl имеется несколько операторов, которые предназначены для использования в числовых выражениях. Некоторые из них вам уже встречались, а с остальными мы сейчас познакомимся. Первой разновидностью уже знакомых вам операторов являются арифметические операторы. В табл. 2.3 приведен их список.

Таблица 2.3. **Арифметические операторы**

Пример	**Название оператора**	**Значение выражения**
5 + $t	Сложение	Сумма 5 и $t
Sy - $x	Вычитание	Разница между $y и $x
$e * $pi	Умножение	Произведение $e на $pi
$f / б	Деление	Частное от деления $f на 6
24 % 5	Остаток от деления	Остаток от деления 24 на 5 (4)
4 ** 2	Возведение в степень	4 в квадрате

Арифметические операторы выполняются в порядке приоритетов, принятых в математике: сначала возведение в степень, затем умножение, деление, вычисление остатка от деления и только затем сложение и вычитание. Если вы не уверены; в каком порядке будут выполняться операции в вашем выражении, используйте скобки. В выражениях с вложенными скобками вначале вычисляются элементы выражения во внутренних скобках:

```
5*6+9;         # Значение 39
5*(6+9);       I Значение 75
5+(6*(4-3));   # Значение 11
```

Строковые операторы

Perl может оперировать не только числами, но и строками. Первым строковым оператором, который мы рассмотрим, является оператор конкатенации (.)• Этот оператор берет строку, находящуюся слева от него, и строку справа и возвращает строку, объединяющую предыдущие две, например:

```
$a="Привет, мой Свет!";
$b=" Как я рад тебя видеть";
$c=$a . $b;
```

В этом примере переменные $a и $Ь имеют простые строковые значения. В последней строке переменной $c присваивается значение Привет, мой Свет! Как я рад тебя видеть, при этом значения переменных $a и $Ь не изменяются.

Конкатенацию строк можно выполнить и другим способом. Раньше мы уже говорили, что внутри строк, заключенных в двойные кавычки, Perl "выискивает" имена переменных. Найденные переменные *интерпретируются*. Это означает, что имя переменной внутри строки в двойных кавычках заменяется реальным значением этой переменной, например:

```
$name="Джон";
print "Привет, $name!";
```

В этом примере Perl ищет в строке в двойных кавычках имена переменных, находит имя $name и подставляет вместо него строку Джон. Этот процесс называется *интерпретацией значения переменных*. Для того чтобы переменные не интерпретировались, нужно заключить строку в одинарные кавычки (тогда Perl вообще никак не будет анализировать эту строку) или поставить обратную косую черту перед идентификатором имени переменной, например:

```
$name="Ringo";
print 'Я использую переменную $name'; # Не будет печатать слово "Ringo"
print "Я использую переменную \$name"; # Также не будет печатать слово "Ringo"
```

Оба оператора print из предыдущего примера напечатают строку Я использую переменную $name, при этом переменная Sname не будет интерпретирована. Итак, оператор конкатенации можно заменить строками в двойных кавычках следующим образом:

```
$fruit1="яблоки";
$fruit2="и груши";
$bowl="$fruit1 $fruit2";
```

Там, где Perl не может четко выделить имя переменной из остальной части строки, заключите имя переменной в фигурные скобки {}. Подобный прием позволяет Perl легко определить имя переменной, например:

```
$name="Thurs";
print "I went to the fair on ${date}day";
```

Без этих скобок непонятно, какую переменную должен интерполировать Perl — $date или $dateday. Фигурные скобки однозначно указывают, что $date.

Следующим строковым оператором является оператор повторения x. В операторе x указываются два аргумента — строка, которую необходимо повторить, и число повторений. Например:

```
$line="-" x 70;
```

В предыдущем примере символ "-" повторяется 70 раз оператором x. Результат сохраняется в переменной $line.

Другие операторы

В Perl существует такое огромное количество операторов, что в этой книге просто не хватит места, чтобы их все подробно описать. Оставшаяся часть этого занятия посвящена часто используемым операторам и функциям Perl.

Унарные операторы

До сих пор все встречавшиеся вам операторы имели по два аргумента. Для деления 6/3 нужен числитель 6 и знаменатель 3, для умножения 5*2 нужны множители 5 и 2 и т.д. Другой тип операторов — операторы с одним аргументом (операндом). Скорее всего, вы уже знакомы с одним представителем операторов такого рода — унарным минусом (-). Унарный минус изменяет знак своего аргумента, например:

```
6;           # Шесть
-6           # Минус шесть
-(-5)        # Пять, минус на минус дает плюс.
```

В отличие от унарного минуса, большинству операторов Perl назначены имена. Операнды *именованных* унарных операторов заключать в скобки необязательно (в табл. 2.4 они приведены лишь для удобства). Поскольку именованные операторы в Perl напоминают функции, то их операнды иногда называют *аргументами,* т.е. термином, принятым для параметров функций Perl.

Краткий список именованных операторов приведен в табл. 2.4.

Таблица 2.4. Некоторые именованные унарные операторы Perl

Оператор	Пример использования	Результат
int	int(5.6234)	Возвращает целую часть аргумента (5)
length	length("nose")	Возвращает длину строки-аргумента (4)
lc	lc("ME TOO")	Переводит все символы строки аргумента в нижний регистр ("me too")
uc	uc("hal 9000")	Действие, обратное действию предыдущего оператора ("HAL 9000")
cos	cos(50)	Косинус 50 радиан (.964966).
rand	rand(5)	Возвращается случайное число в диапазоне от 0 до указанного в аргументе; если аргумент отсутствует — возвращается случайное число в диапазоне от 0 до 1

Полный список именованных операторов приведен в документации по Perl. На 1-м занятии, "Начало работы с Perl", рассказывалось о том, как получить доступ к разделам документации с помощью утилиты perldoc, которая входит в поставку Perl. Полный список операторов находится на страницах руководства perlop и perlfunc. На последующих занятиях, по мере необходимости, будут представлены и некоторые другие операторы.

Инкремент и декремент

В разделе "Числовые операторы" мы уже говорили о специальном типе операторов присваивания, называемом *инкрементом,* выглядевшем тогда так:

```
$counter=$counter+1;
```

Инкремент обычно используется для подсчета чего-либо, например количества встретившихся записей или для генерации последовательных номеров (например, нумерации элементов списка). Вы можете в этих целях использовать специальный опе-

ратор, называемый *оператором автоинкремента* (++). Этот оператор прибавляет к операнду единицу:

```
$counter++;
```

После выполнения этой строки кода значение переменной `$counter` увеличивается на единицу.

В Perl есть оператор для уменьшения значения переменной, который, как вы уже могли сами догадаться, называется *оператором автодекремента* (--). Автодекремент используется точно так же, как и автоинкремент:

```
$countdown=10;
$countdpwn--;  # Значение переменной становится равным 9
```

Перед тем как завершить описание этих операторов, следует упомянуть еше об одном любопытном свойстве автоинкремента. Речь идет о применении этого оператора к строке алфавитно-цифровых символов, что приводит к очень интересному результату! Данная операция влияет на последний (самый правый) символ строки — его значение увеличивается на 1, при этом буквенный символ заменяется следующим символом алфавита. Ниже приведены примеры действия оператора автоинкремента на строки.

```
$a="999";
$a++;
print $a;    # как н следовало ожидать, напечатает 1000.

$a="c9";
$a++;
print $a;    # напечатает d0. 9+1=10, с заменяется следующим
             \ символом алфавита d.

$a="zzz";
$a++;
print $a;    t напечатает aaaa.
```



Оператор автодекремента не выполняет подобных действий.

УГЛОВОЙ оператор

Угловой оператор (о), иногда из-за своей формы называемый "бубновым", прежде всего, используется для чтения и записи файлов (подробнее об этом речь пойдет на 5-м занятии, "Работа с файлами"). Однако, чтобы сделать примеры более интересными, мы начнем его использовать раньше, и к 5-му занятию он уже будет нам хорошо знаком.

А пока мы будем использовать простейшую форму углового оператора: <STDIN>. Эта конструкция сообщает Perl, что строка должна быть считана со стандартного устройства ввода — обычно с клавиатуры. Таким образом, оператор <STDIN> возвращает строку, введенную с клавиатуры, например:

```
print "Какой у вас размер обуви? ";
$size=<STDIN>;
print "Ваш размер обуви ~ ${size}Спасибо за внимание";
```

После выполнения этого кода (предположим, что в качестве размера обуви вы указали число 45) на экран будет выведено следующее:

```
Какой у вас размер обуви? 45
Ваш размер обуви — 45
Спасибо за внимание
```

Оператор <STDIN> читает вводимые с клавиатуры символы до тех пор, пока пользователь не нажмет клавишу <Enter>. Затем введенная строка помещается в переменную Ssize. Строка текста, возвращаемая оператором <STDIN>, содержит символ перевода строки, введенный пользователем при нажатии клавиши <Enter>. Обычно не желательно, чтобы в конце введенной строки находился символ перевода строки, т.е. требуемая строка должна содержать лишь текст. Для удаления символа перевода строки можно воспользоваться оператором chomp:

```
print "Какой у вас размер обуви? ";
$size=<STDIN>;
chomp $size;
print "Ваш размер обуви — $size. Спасибо за внимание\n";
```

Оператор chomp удаляет в строке-аргументе завершающий символ перевода строки. Он также возвращает количество удаленных символов — обычно это 1, но иногда О, если ничего не удалено.

Другие операторы присваивания

Вы уже знаете, что для присваивания значений скалярным переменным используется оператор присваивания =. В Perl имеется еще целый набор операторов присваивания. Каждый арифметический оператор Perl и еще некоторые другие могут быть использованы одновременно и для выполнения соответствующей операции, и для последующего присваивания. Ниже приведено общее правило образования таких операторов присваивания:

переменная **оператор=** *выражение*

Эта строка равносильна следующей:

переменная=переменная *оператор выражение*

Использование подобных операторов обычно не сделает вашу программу более читабельной, но сделает ее короче. Согласно вышеописанному правилу, оператор

```
$a=$a+3;
```

можно сократить до

```
$a+=3;
```

А вот еще несколько примеров:

```
$line.=" - это конец строки";  I Фраза дописывается к $line;
$y*=$x   # То же, что и $y=$y*$x
$r%=67   # Остаток от деления $r на 67 помещается в $r
```

Несколько слов о строках и числах

Perl позволяет использовать в выражениях как строки, так и числа. При этом, в зависимости от ситуации, Perl выполняет автоматическое преобразование чисел в строковое представление и наоборот. Ниже приведено несколько правил, которыми руководствуется программа-интерпретатор.

- Если из строки можно без проблем выщелить число, Perl использует число, например:

```
$a=42;          # Число
print $a+18;    # Печатает 60
$b="50";        # Строка
print $b-10;    # Печатает 40
```

- Если число используется в строковом выражении, Perl преобразовывает число в строку, например:

```
$a=42/3;          # Число
$a=$a . "Hello"; # Используется число как строка
Print $a;         # Печатает "14Hello"
```

- Если в выражении строка используется там, где должно быть число, Perl использует число 0, например:

```
$a="Hello, World!";
print $a+6;  # Печатает число 6
```

  Однако, если включен режим вывода предупреждений, в последнем случае Perl выдаст соответствующее сообщение.

Все эти примеры иллюстрируют философию, свойственную Perl, — философию "минимальной неожиданности". Даже получив абсурдные аргументы, как в предыдущем примере, Perl пытается сделать с ними нечто осмысленное. Если вы включили режим вывода предупреждений, указав ключ -v в первой строке вашей программы или в командной строке, Perl предупредит вас, что совершает бессмысленное действие, следующим сообщением: Argument X isn't numeric.

Упражнение: вычисление сложных процентов

ДЛЯ ЭТОГО упражнения мы выбрали задачу вычисления сложных процентов. Программа будет высчитывать доход, исходя из информации о процентных ставках, сумме депозита и времени хранения. Мы будем при этом использовать следующую формулу:

$$Сумма = Взнос\left(\frac{(1+Процентная_ставка)^{Срок_хранения}-1}{Процентная_ставка}\right)$$

Наберите в текстовом редакторе программу, приведенную в листинге 2.2, и назовите ее interest. Номера строк набирать не нужно. Сделайте файл interest исполняемым, следуя инструкциям, приведенным ранее в главе.

После этого попытайтесь запустить программу, набрав в командной строке

```
perl interest
```

В листинге 2.3 приведен пример вывода программы interest.

Листинг 2.2. Полный исходный код программы interest

```
#!/usr/bin/perl -v

print "Величина месячного взноса? ";
$pmt=<STDIN>;
chomp $pmt;

```

```
7: print "Годовая процентная ставка? (пример 7% — .07) ";
8: $interest=<STDIN>;
9: chomp $interest;
10:
11:print "Период депозита в месяцах? ";
12:$mons=<STDIN>;
13:chomp $mons;
14:
15:# В формуле заложена месячная процентная ставка
16:$interest/=12;
17:
18:$total=$pmt * ( ( 1 + $interest) ** ($mons -1) )/ $interest;
19:
20:print "После $mons месяцев при ежемесячной ставке $interest \n";
21:print "у вас будет сумма $total.\n";
```

Проведем анализ программы.

- *Строка 1.* В этой строке находятся имя программы интерпретатора (вы можете изменить его в соответствии с конфигурацией вашей системы) и ключ `-w`. Всегда включайте режим выщачи предупреждений!
- *Строка 3.* Пользователь вводит размер месячного взноса.
- *Строка 4.* Значение переменной $pmt считывается со стандартного устройства ввода (клавиатуры).
- *Строка 5.* Удаляется символ перевода строки в конце `$pmt`.
- *Строки 7~9.* Считывается с клавиатуры значение переменной `$interest` и удаляется символ перевода строки.
- *Строки 11—13.* Считывается с клавиатуры значение переменной `$mons` и удаляется символ перевода строки.
- *Строка 16.* Содержимое переменной $interest делится на 12, результат помешается в переменную $interest.
- *Строка 18.* Вычисляются сложные проценты, результат заносится в переменную `$total`.
- *Строки 20—21.* Печать результатов.

Листинг 2.3, Пример работы программы interest

```
Величина месячного взноса? 180
Годовая процентная ставка? (пример 7% — .07) .07
Период депозита в месяцах? 120
После 120 месяцев при ежемесячной ставке 0.00583333333333333
у вас будет сумма 61652.767054031.
```

Резюме

На этом занятии вы узнали, что основным типом данных в Perl является скаляр. Скалярные переменные могут содержать любые данные. Числовые литералы могут быть представлены в разных форматах — целочисленном, вещественном и в формате с плавающей точкой. Строковые литералы заключаются в двойные или одинарные кавычки. В Perl предусмотрены операторы, позволяющие выполнять строковые и арифметические операции.

Вопросы и ответы

Вывод программы interest выглядит неряшливо. Как можно указать, сколько десятичных знаков нужно выводить?

Проще всего управлять количеством выводимых десятичных знаков с помощью функции printf(), описанной на 9-м занятии, "Дополнительные функции и операторы".

Имеется ли в Perl функция для округления?

Функция printf() округляет числа при выводе. Если вам необходима функция round{), воспользуйтесь модулем POSIX, содержащим как эту функцию, так и многие другие функции.

Какое минимальное и максимальное значение числа допускается в Perl?

Ответ на данный вопрос зависит от того, какая у вас операционная система. В числах двойной точности с плавающей точкой в типичной Intel-совместимой UNIX-системе можно использовать более чем 300-значные числа. Обычно для вычислений вполне достаточно 14 разрядов.

Семинар

Контрольные вопросы

1. Внутри оператора qq переменные интерпретируются:

   а) да;

   б) нет.

2. Определите значение переменной $c после выполнения следующего фрагмента кода:

```
$a=6;
$a++;
$b=$a;
$b--;
$c=$b;
```

   а) 6;

   б) 7;

   в) 8.

3. Конкатенация может быть выполнена лишь с помощью оператора конкатенации (.):

   а) да;

   б) нет.

Ответы

1. Правильный ответ — вариант а). Действие оператора qq аналогично действию двойных кавычек. Это означает, что переменные интерпретируются внутри этого оператора.

2. Правильным будет вариант а). Вначале переменной $a присваивается значение 6, затем увеличивается до 7 и присваивается переменной $Ь. Затем значение переменной $b уменьшается до 6 и присваивается $c.

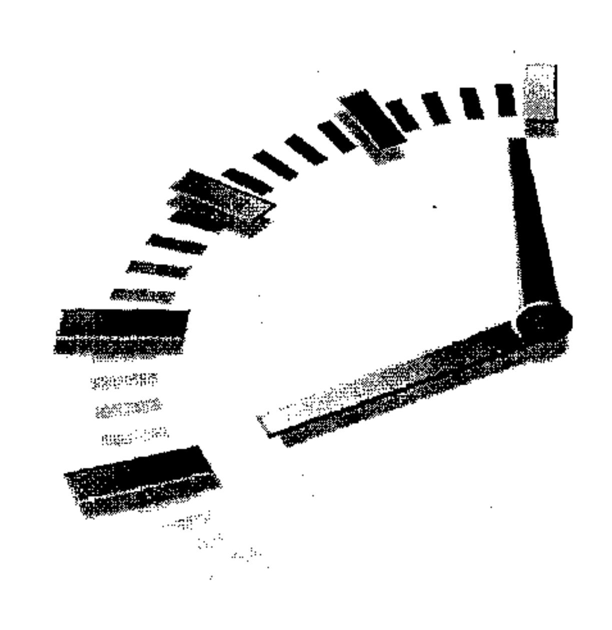
3. Правильный ответ — вариант б). В Perl любое действие можно выполнить несколькими способами. Конкатенацию двух или более скаляров можно произвести, заключив их имена в двойные кавычки:

   qq(ab$c);

Упражнения

- Напишите короткую программу, запрашивающую значение температуры по Фаренгейту и переводящую ее в температуру по Цельсию. Для перевода температуры по Фаренгейту в шкалу Цельсия нужно вычесть из температуры 32 и умножить полученное значение на 5/9. Например, 75 градусов по Фаренгейту соответствуют 21,1 градуса по Цельсию.

- Модифицируйте программу interest таким образом, чтобы выводимые результаты содержали не более двух десятичных знаков. Этого можно добиться без printf(), лишь с помощью умножения, деления и оператора int.



3-й час

Управление процессом выполнения программы

На 2-м занятии, "Строительные блоки Perl: числа и строки", речь шла об операторах и выражениях. Для всех примеров этого занятия характерным было то, что операторы в них выполнялись последовательно друг за другом и только один раз.

Одним из важнейших достоинств компьютеров является возможность автоматизации повторяющихся задач, что освобождает пользователей от выполнения утомительных рутинных операций. До сих пор мы не знали, как заставить Perl выполнить некоторую операцию несколько раз. На этом занятии вы познакомитесь с управляющими структурами Perl, используя которые, вы сможете группировать операторы в так называемые *блоки,* а затем многократно выполнять их необходимое количество раз.

Другим достоинством компьютера является возможность быстрого принятия решений. Было бы очень неудобно, если бы компьютер при необходимости принять какое-либо решение обязательно запрашивал команду пользователя. Во время обычного процесса получения и чтения сообщения электронной почты компьютер без вашего непосредственного участия должен принять огромное количество решений: как объединить фрагменты сетевого трафика, определить цвет каждого пикселя на мониторе, как прочитать и отобразить сообщение, что делать при изменении положения указателя мыши, и бесчисленное множество других. Многие из этих решений влекут за собой принятие других решений, причем некоторые из них нужно принимать тысячи раз в секунду. На этом занятии мы расскажем об *условных операторах.* С их помощью можно создать блоки кода, которые будут выполняться в зависимости от решения, принятого программой.

Основные темы этого занятия.

- Блочные структуры.
- Операторы.
- Циклы.
- Метки.
- Выход из Perl после выполнения программы.

Блоки

Простейшим способом объединения нескольких операторов Perl является *блок*. Для образования блока достаточно заключить группу операторов в фигурные скобки:

```
{
        оператор_а;
        оператор_б;
        оператор_в;
}
```

Внутри блока операторы выполняются последовательно, как и раньше. Блоки могут состоять из других блоков, например:

```
{
        оператор_а;
        {
        оператор_б;
        оператор_в;
        }
}
```

Для Perl вообще и для блоков, в частности, характерно свободное форматирование. Другими словами, операторы и фигурные скобки могут находиться в одной или различных строках. Допустим любой тип выравнивания, единственное условие — чтобы все фигурные скобки имели пару, например:

```
{ оператор;     { другой_оператор; }
        { последний_оператор;         }  }
```

Несмотря на то что вы можете располагать код в блоках как вам заблагорассудится, беспорядочное нагромождение операторов затрудняет чтение программы. Необязательные, но желательные отступы делают программу Perl удобочитаемой.

Самостоятельные блоки в программе Perl называются *изолированными*. В большинстве же случаев блоки находятся в других операторах Perl.

Оператор if

Для управления условным выполнением операторов Perl обычно используется оператор if. Синтаксис этого оператора приведен ниже:

```
if (выражение) БЛОК
```

Работает оператор if так: если *выражение* истинно, блок кода выполняется. Если выражение ложно, блок кода не выполняется. Рассмотрим пример:

```
if ( $r == 5 ) {
    print 'Переменная $r равна 5.';
}
```

В этом примере проверяется значение переменной `$r` на равенство 5. (Символы == — это оператор равенства; не путайте его с оператором присваивания =.) Если операнды с обеих сторон этого оператора (в нашем случае `$r` и 5) численно равны, выражение считается истинным и выполняется оператор `print`. Если значение `$r` не равно 5, оператор `print` не выполняется.

Оператор if позволяет также выполнить один фрагмент кода, если некоторое условие истинно, и другой фрагмент кода — если это условие ложно. Это достигается с помощью использования оператора `if-else`. Его синтаксис имеет вид

```
if (выражение)  # Если выражение истинно...
    БЛОК        # .выполняется этот блок кода.
  Else
    БЛОК        # В противном случае выполняется этот блок.
```

Блок, следующий за выражением, выполняется, если оно истинно, а блок после ключевого слова else — если выражение ложно. Приведем пример использования описываемого оператора:

```
$r=<STDIN>; chomp $r;
if ($r == 10) {
    print '$r равно 10';
} else {
    print '$r не равно 10...';
    $r=10;
    print '$r присвоено значение 10';
}
```



> Обратите внимание, что в предыдущем примере для присваивания значения переменной `$r` используется оператор =. Для проверки значения `$r` используется оператор равенства ==. Это два совершенно разных оператора с абсолютно различным действием. Не путайте их, иначе отладка ваших программ может сильно усложниться. Помните, что с помощью оператора = значения присваиваются переменным, а с помощью оператора == — выражение проверяется на равенство определенному значению.

Существует еще одна разновидность оператора if, с помощью которой можно проверить значения нескольких выражений и выполнить код, соответствующий истинному условию:

```
if (выражение!)         # Если выражение 1 истинно...
    БЛОК1               # ..выполняется этот блок кода.
elseif (выражение2)     # Иначе, если выражение2 истинно...
    БЛОК2               # ..выполняется этот блок кода.
 else
    БЛОК3               # Если ни одно из выражений не истинно,
                        #   выполняется этот блок.
```

Данный оператор выполняется следующим образом: если выражение, помеченное как *выражение!*, истинно — выполняется блок кода *БЛОК!*. Иначе управление передается оператору elseif и проверяется значение *выражения2*. Если ни *выражение1*, ни *выражение2* не являются истинными, выполняется *БЛОК3*. Для демонстрации такой синтаксической конструкции рассмотрим реальный фрагмент программы на Perl:

```
$r=10;
if ($r == 10) {
   print '$r равно 10!';
} elseif ($r == 20) {
   print '$r равно 20!';
} else {
   print '$r не равно ни 10, ни 20';
}
```

Другие операторы отношения

До сих пор мы сравнивали значения лишь с помощью оператора равенства ==. В Perl имеется еще ряд операторов, предназначенных для сравнения численных значений, большинство из них представлены в табл. 3.1.

Таблица 3.1. Числовые операторы отношения

Оператор	Обозначение	Пример	Описание
Равенства	==	$x==$y	Истина, если $x равно $y
Больше чем	>	?x>$y	Истина, если $x больше $y
Меньше чем	<	$x<$y	Истина, если $x меньше $y
Больше или равно	>=	$x >= $y	Истина, если $x больше или равно $y
Меньше или равно	<=	$x <= $y	Истина, если $x меньше или равно $y
Не равно	!=	$x != $y	Истина, если $x не равно $y

Эти операторы могут использоваться в любом месте программы для сравнения численных значений, например в операторе if, как показано в листинге 3.1.

Листинг 3.1. Игра в угадайку

```
1: #!/usr/bin/perl -w
2:
3:  $im_thinking_of=int(rand10);
4: print "Введите число от 0 до 9:";
5: $guess=<STDIN>;
6: chomp $guess;  i Ве забудьте удалить символ новой строка!
7:
8: if ($guess>$im_thinking_of) {
Э:      print "Перебор!\n";
10: } elsif ($guess < $im_thinking_of) {
11:             print "Недобор!\n";
12: } else {
13:              print "Вы угадали!\n";
14:}
```

 Проведем анализ программы.

- *Строка 1.* Это стандартная первая строка программ на Perl. В ней указывается Полный путь к программе-интерпретатору, а ключ -w активизирует режим выдачи предупреждений. На 1-м занятии, "Начало работы с Perl", мы говорили о том, что в каждом конкретном случае эта строка может несколько видоизменяться.
- *Строка 3.* Функция rand 10 генерирует случайное число в диапазоне от 0 до 10, а оператор int() возвращает его целую часть. Затем получившееся целое число от 0 до 9 присваивается переменной `$im_thinking_of`.
- *Строки 4–6.* В этом месте программы у пользователя запрашивается значение, которое присваивается переменной $guess. С помощью функции chomp завершающий символ перевода строки удаляется.

- *Строки 8—9.* Если значение переменной $guess больше, чем переменной $im_thinking_of, выводится соответствующее сообщение.
- *Строки 10—11.* Если же значение переменной $guess меньше, чем переменной $im_thinking_of, выводится другое сообщение.
- *Строки 12—13.* Оставшийся вариант — пользователь угадал число.

Операторы в табл. 3.1 используются для сравнения лишь численных значений. Их использование для сравнения нечисловых данных приводит к неожиданному результату. Рассмотрим пример:

```
$first="Simon";
$last="simple";
if ($first == $last) { t == - совсем не то, что вы ожидали!
    Print "Значения переменных равны!\n";
}
```

В результате выполнения этого фрагмента кода окажется, что значения переменных $first и $last численно равны. На 1-м занятии, "Начало работы с Perl", говорилось, что если нечисловые строки используются там, где Perl ожидает встретить число, то вместо них подставляется нулевое значение. Поэтому предшествующее выражение оператора if в Perl равносильно следующему: if { 0 == 0). Это выражение истинно, и результат действия оператора if совсем не такой, как вы, вероятно, ожидали.



Если режим вывода предупреждений включен, то сравнение двух нечисловых значений (в предыдущем примере это Simon и simple) с помощью оператора == приведет к появлению соответствующего сообщения.

Для сравнения нечисловых значений используйте другие операторы Perl, приведенные в табл. 3.2.

Таблица 3.2. Строковые операторы отношения

Оператор	Обозначение	Пример	Описание
Равенства	eq	$s eq $t	Истина, если строка в переменной $s совпадает со строкой $t
Больше чем	gt	$s gt $t	Истина, если строка в переменной Ss больше значения переменной $t
Меньше чем	lt	$s lt$t	Истина, если строка в переменной Ss меньше значения переменной $t
Больше или равно	ge	$s ge $t	Истина, если строка в переменной $s больше или равна значению переменной $t
Меньше или равно	1e	$s le $t	Истина, если строка в переменной $s меньше или равна значению переменной $t
Не равно	пе	$s ne $t	Истина, если строка в переменной $s не равна строке $t

Данные операторы выполняют анализ текстовых строк на основе сравнения ASCII-кодов соответствующих символов, начиная с первого. Это означает, что строки, расположенные первыми в алфавитном порядке, имеют высший приоритет. Таким образом, наибольший приоритет имеют знаки пунктуации, затем числа, прописные буквы и, наконец, строчные буквы. Например, строка 1506 больше строки Happy, а та, в свою очередь, больше строки happy.

Что есть Истина в Perl

До сих пор мы свободно пользовались термином *истины:* "если это выражение истинно...", не имея его формального определения. В Perl есть несколько коротких правил относительно того, что истинно, а что ложно. Правила эти таковы:

- число 0 имеет значение Ложь;
- пустая строка ("") или строка "0" имеет значение Ложь;
- неопределенные значения undef имеют значение Ложь;
- все остальные значения считаются истинными.

Логично, не правда ли? Единственно, о чем еще следует помнить — перед тем, как проверить на истинность некоторое выражение, его необходимо упростить: определить значения вызываемых функций и арифметических выражений. Затем полученное значение выражения нужно перевести в скалярный вид и только тогда решить, истинно оно или ложно.

Постарайтесь осмыслить эти правила и посмотрите табл. 3.3. Попробуйте сначала сами определить, какое выражение истинно, а какое ложно.

Таблица 3.3. Примеры истинных и ложных выражений

Выражение	Истина или Ложь?
0	Ложь. Число 0 имеет значение Ложь
10	Истина. Ненулевое число
9>8	Истина. Операторы отношения возвращают значения Истина или Ложь, как и следует ожидать
-5+5	Ложь. Значение этого выражения — число 0, имеющее значение Ложь
0.00	Ложь. Это число — другое представление числа 0, как и 0x0, 00, 0b0 и 0e00
""	Ложь. Этот случай явно указан в правилах
" "	Истина. Так как в кавычках находится пробел, строка считается непустой
"0.00"	Истина. Удивлены? Это уже строка, причем не "" или "0"
"00"	Истина. По тем же причинам
"0.00"+0	Ложь. Сначала вычисляется значение выражения, которое равно 0

Пока в операторе if мы использовали только операторы отношения, хотя, в принципе, можно использовать *любое* выражение, которое в конечном счете будет приводиться к значению Истина или Ложь.

```
# Анализируется значение скалярной переменной $a
# и определяется его логическое значение
if ($a) { ... }
```

```
# Вычисляется длина переменной $b. Если она ненулевая,
# логическое значение выражения - Истина.
If (length($b)) { .... }
```

В Perl имеется специальное значение — undef. Оно присваивается всем неинициализированным переменным. Кроме того, некоторые функции в случае неудачного исхода их выполнения возвращают это значение. Значение undef трактуется интерпретатором Perl как *неопределенное.* Оно не равно нулю или любому значению в привычном понимании этого слова. При проверке на истинность, например в операторе if, undef всегда имеет логическое значение Ложь. В арифметических выражениях вместо undef всегда подставляется 0.



Использование неинициализированных переменных обычно свидетельствует об ошибке в программе. Если в программе на Perl с включенным режимом вывода предупреждений значение undef используется в выражениях или передается в виде аргумента функциям — появляется сообщение Use of uninitialized value.

Логические операторы

Допустим, вам требуется написать код, выполняющий что-либо, если переменные $x и $y имеют истинное значение, а $z — ложное. Такой код можно реализовать с помощью плохо читаемого набора операторов if:

```
if ($x) {
   if ($y) {
      if ($z) {
      # Ничего не происходит
   } else {
            print "Нужное условие достигнуто. \n";
            }
         }
      }
```

В Perl имеется целый класс операторов, предназначенных для объединения логических выражений. Это — так называемые *логические операторы,* которые описаны в табл. 3.4.

Таблица 3.4. Логические операторы

Оператор	Обозначение	Альтернативное обозначение	Пример	Описание
Логическое И	&&	and	$s && $t	Истина, только если $s и $t имеют истинное значение
			$q and $p	Истина, только если $q и $p имеют истинное значение
Логическое ИЛИ	||	or	$s || $t	Истина, если $s или $t имеют истинное значение

Окончание табл. 3.4

Оператор	Обозначение	Альтернативное обозначение	Пример	Описание
			$q or $p	Истина, если $д или $p имеют истинное значение
Отрицание	!	not	! $m	Истина, если $m имеет ложное значение
			not $m	Истина, если $т имеет ложное значение

Предыдущий пример можно переписать с помощью операторов из табл. 3.4 следующим образом:

```
if ($x and $y and not $z) {
    print "Нужное условие достигнуто. \п";
}
```

Выражения, объединенные логическими операторами, вычисляются слева направо до тех пор, пока не появится возможность заранее определить значение всего логического выражения (листинг 3.2).

Листинг 3.2. Примеры использования логических выражений

```
1: $a=0; $b=1; $c=2; $d="";
2:
3: if {$a and $b) { print 'Переменные $a и $b истинны'; }
4: if ($d or $b)  { print 'Хотя бы одна из переменных $d или $Ь истинна'; }
5: if {$d or not $b or $c)
6:                { print 'Переменная $d истинна, или $Ь ложна, или $c истинна';
   }
```

 Проведем анализ программы.

- *Строка 1.* В этой строке переменным присваиваются значения.
- *Строка 3.* Вначале определяется логическое значение переменной $a. Поскольку оно ложно, выражение с использованием оператора and никогда не может быть истинно. Поэтому логическое значение переменной $b не определяется — в этом нет необходимости, ведь уже известно, что выражение ложно. В результате оператор print не выполняется.
- *Строка 4.* Вначале определяется логическое значение переменной $d. Оно ложно, но все выражение с оператором or все еще может быть истинным, поэтому вычисляется логическое значение переменной $Ь, оно истинно, значит, и все выражение истинно, следовательно, оператор print выполняется.
- *Строка 5.* Вначале определяется логическое значение переменной $d. Оно ложно, но, несмотря на это все, выражение может быть истинным по той же причине, что и в строке 4, — в нем содержится логический оператор or. Зна-

чение переменной $b истинно, следовательно выражение not $b — ложно. Истинность или ложность всего выражения еще не установлена, осталось проверить значение переменной $c. Эта переменная имеет истинное значение, следовательно, и все выражение истинно, и оператор print выполняется.

Выше была описана особенность определения значения логических выражений, которая состоит в том, что при первой же возможности установить истинность всего выражения, дальнейшие вычисления не производятся. Эта особенность имеет специальное название — *короткозамкнутость* логических выражений. Она позволяет программистам на Perl конструировать простые операторы управления процессом выполнения программы без логических операторов и даже вообще без оператора if:

```
$message='Переменные $a и $b истинны.';
($a and $b) or $message="Одна или обе переменные $a или $Ь ложны.";
```

В предыдущем примере, если одна из переменных $a или $b имеет значение Ложь, будет вычисляться правая часть оператора or, и значение переменной $message модифицировано. Если же обе переменные истинны, значение выражения or также будет истинно, поэтому правая часть оператора or не выполняется. В результате значение переменной $message не меняется. В этом примере использован побочный эффект короткозамкнутости операторов or и and для изменения значения переменной $message.



Строго говоря, операторы [| и or не эквивалентны. Отличие состоит в том, что оператор | [имеет более высокий приоритет, чем or. Это означает, что при прочих равных условиях в выражениях в первую очередь будут выполняться операторы ||, аналогично тому, как в арифметических выражениях операции умножения выполняются перед операциями сложения. Данное замечание относится и к парам &&/and, и !/not. Однако для большей надежности используйте скобки, гарантирующие требуемый порядок вычисления выражения.

Интересно то, что логические операторы Perl не просто возвращают значение Истина или Ложь. Они возвращают последнее вычисленное значение. Например, выражение 5 && 7 возвращает значение 7 (оно является истинным), поскольку в данном случае оно было вычислено последним. Такое свойство логических выражений позволяет использовать приведенные ниже конструкции.

```
# Переменной $nev присваивается старое значение,
#  в случае истинности значения переменной $old.
f в противном случае присваивается строка "default".
$new=$old || "default";
```

Это выражение выглядит намного изящнее, чем следующее:

```
$new=$old;
if (! $old) { # Если значение $old не определено или ложно?
      $new="default";
}
```

Циклы

В начале этой главы говорилось о том, что принятие решений и условное выполнение кода — не все, что может понадобиться от программы. Иногда еще нужно многократно выполнять некоторый фрагмент кода. Выше, в листинге 3.1, мы рас-

сматривали пример игровой программы угадывания чисел. Было бы неплохо ее усовершенствовать и дать пользователю несколько попыток. Для этого необходимо реализовать условный повтор определенных фрагментов кода, что легко можно сделать с помощью циклов.

Организация циклов с оператором while

Циклы while считаются наиболее простыми. Оператор while повторяет блок кода до тех пор, пока некоторое выражение истинно. Вот синтаксис этого оператора:

while *(выражение) БЛОК*

Когда интерпретатор Perl встречает оператор while, проверяется выполнение условия. Если *выражение* истинно, выполняется *БЛОК* кода. После выполнения всего блока повторно вычисляется значение *выражения,* если оно истинно, блок повторяется (листинг 3.3).

Листинг 3.3. Пример цикла while

```
1: $counter=0;
2: while ($counter < 10 ) {
3:     print "Выполняется $counter итерация цикла\n";
4:     $counter++;
5: }
```

 Проведем анализ программы.

- *Строка 1.* Переменной `$counter` присваивается нулевое значение.
- *Строка 2.* Вычисляется значение выражения `$counter` < 10. Если оно истинно, выполняется блок кода.
- *Строка 4.* Значение переменной `$counter` увеличивается на единицу.
- *Строка 5.* Фигурная скобка } закрывает блок, начинающийся в строке 2 с {. В этот момент Perl возвращается в начало цикла while и заново вычисляет выражение в круглых скобках.

Организация циклов с оператором for

Оператор for — более сложная, но в то же время более универсальная конструкция для организации циклов в Perl. Его синтаксис выглядит так:

for *(инициализация; условие; инкремент) БЛОК*

Три раздела оператора for, *инициализация, условие* и *инкремент,* разделены точками с запятой. Когда Perl встречает оператор for, выполняется следующая последовательность действий.

- Вычисляется выражение *инициализации.*
- Вычисляется выражение, задающее условие окончания цикла. Если оно истинно — выполняется *БЛОК* кода.

- После выполнения блока производится приращение счетчика и снова проверяется условие. Если оно по-прежнему истинно, блок кода выполняется повторно. Этот процесс продолжается до тех пор, пока не перестает выполняться условие.

Ниже приведен пример цикла for:

```
for ( $a=0; $a<10; $a=$a+2 J {
    print "A равно $a\n";
}
```

Здесь переменной $a присваивается значение 0, затем выполняется проверка условия $a<10, оказавшегося истинным. В теле цикла выводится сообщение. Затем значение переменной $a увеличивается на 2: $a=$a+2. Снова выполняется проверка и цикл. Это продолжается до тех пор, пока $a станет равно 10. В этом случае проверочное условие ложно, и программа выходит из цикла.

Использовать счетчик в операторе for необязательно, цикл будет выполняться до тех пор, пока истинно выражение, задающее условие окончания цикла. Более того, все три части оператора for необязательны, необходимо лишь присутствие двух символов точки с запятой. Например, этот оператор for прекрасно обходится без разделов инициализации и инкремента:

```
$i=10;                # Инициализация значения переменной цикла
for { ; $i>-1; ) {
    print "$i..";
    $i--;             # Декремент.
}
print "Цикл окончен!\n";
```

Другие средства управления программой

Управлять процессом выполнения программы с помощью циклов и условных операторов достаточно удобно, но для создания программ с прозрачной структурой нужны и другие операторы управления. Действительно, в Perl имеются операторы для досрочного завершения цикла while, для пропуска определенной части цикла for, для выхода из блока условного оператора if и для выхода из программы вообще. Некоторые из перечисленных конструкций рассмотрены в этом разделе. Их использование может облегчить чтение программы.

Альтернативная запись оператора if

Оператор if может иметь другой синтаксис. Если внутри блока имеется только одно выражение, оно может предшествовать if. Так, вместо кода

if **(условное_выражение)** *{выражение;}*

можно написать:

выражение if *(условное_выражение);*

Ниже приведено несколько примеров подобного синтаксиса:

```
$correct=1 if ($guess == $question);
print "Это не число pi!" if ( $ratio != 3.14159);
```

Этот синтаксис в Perl используется для ясности, поскольку обычно легче читать код, в котором действие предшествует условию. Перед оператором if может быть только одно выражение, а в конце оператора обязательно должна быть точка с запятой.

Операторы управления циклами

Кроме блоков, операторов for, while, if, управляющих порядком выполнения блоков, в Perl имеются операторы для управления программой *внутри* самих блоков.

Одним из таких операторов является last. С его помощью можно выйти из внутреннего выполняемого блока цикла. Вот пример:

```
while ($i<15) {
     last if ($i==5);
     $i++;
}
```

Оператор last позволяет досрочно завершить выполнение цикла while, если значение переменной $i равно 5. При этом не нужно дожидаться, пока условное выражение примет ложное значение. В случае использования вложенных циклов оператор last завершает выполнение текущего внутреннего цикла.

Приведенная в листинге 3.4 программа находит все пары чисел меньше 100, произведение которых равно 140, например 2 и 70, 4 и 35 и т. д. Делается это крайне неэффективно, но нас будет интересовать в основном использование оператора last. Как только очередная пара найдена и выведена на экран, программа выходит из внутреннего цикла с итерацией по $j. При этом внешний цикл (с инкрементом $i) продолжает выполняться и снова запускает внутренний цикл.

Листинг 3,4. Нахождение пар чисел

```
for($i=0; $i<100; $i++) {
   for($j=0; $j<100; $j++) {
     if <$i * $j == 140) {
        print "Произведение $i и $j равно 140\n";
          last;
     }
   }
}
```

Кроме last, в Perl существует также оператор next. Он завершает текущую итерацию цикла и передает управление в начало цикла, например:

```
for <$i=0; $i<100; $i++) {
   next if (not $i % 2);
   print "Число =$i нечетно\n";
}
```

Этот цикл выводит все нечетные числа в диапазоне от 0 до 100. Оператор next запускает следующую итерацию цикла, если $i содержит четное число. Выражение $i % 2 возвращает остаток от деления $i на 2, а оператор not инвертирует полученное логическое значение. Таким образом, если число четное, оператор print пропускается. Естественно, что существуют более удачные алгоритмы поиска нечетных чисел, но тогда мне пришлось бы придумать другой пример для иллюстрации оператора next.

Метки

Perl позволяет помечать блоки и некоторые операторы циклов (for или while). Для этого перед блоком или оператором помешают специальный идентификатор, который называется *меткой,* например:

```
MYBLOCK: {
}
```

Предыдущий блок помечен как MYBLOCK. Имена меток следуют тем же правилам, что и имена переменных, за небольшим исключением: в отличие от имен переменных, метки не должны иметь символов наподобие %, $, @. Важно также, чтобы имена меток не совпадали с зарезервированными словами Perl. Хорошим стилем является использование в именах меток только прописных букв. Это позволит избежать конфликтов имен с настоящими и будущими служебными словами Perl. Операторы for и while также могут быть помечены, например:

```
OUTER: while ($expr ) {
       INNER: while ($expr) {
            операторы;
       }
}
```

Метку можно указывать в качестве аргумента в таких операторах, как last и next. Это позволяет досрочно завершить выполнение любого блока операторов. В листинге 3.4 мы находили пары чисел, произведение которых равно 140, с использованием вложенных циклов for. А теперь представьте, что нам нужно найти только первую пару множителей. Чтобы выйти из обоих циклов после нахождения результата, потребуется сложная комбинация переменных-флагов и условных операторов. И все это только для того, чтобы закончить выполнение внешнего цикла из внутреннего! Однако данную задачу можно решить значительно проще. Судите сами:

```
OUTER: for($i=0; $i<100; $i++) {
        for($j=0; $j<100; $j++) {
          if ($i * $j == 140) {
            print "Произведение $i и $j равно 140\n";
            last OUTER;
        }
    }
}
```

Теперь оператору last явно указан цикл, из которого нужно выйти, — это цикл, помеченный как OUTER. В результате данная программа напечатает только первую пару найденных ею множителей, произведение которых равно 140.

Выход из Perl

Оператор exit — одно из радикальных средств управления программой. Как только Perl встречает этот оператор, программа перестает выполняться и управление возвращается операционной системе вместе со специальным *кодом завершения.* Значение этого кода определяет, успешно была завершена программа или нет. Более подробно коды завершения будут рассмотрены на 11-м занятии, "Взаимодействие с операционной системой". А пока достаточно знать, что код 0 означает, что все в порядке. Вот пример использования оператора exit:

```
if ($user_response eq 'выйти') {
  print "Завершение работы программы.\п";
  exit 0;
}
```



Оператор exit обладает важным побочным действием, относящимся к действиям операционной системы. После его выполнения все открытые программой файлы закрываются, освобождается выделенная Perl память и интерпретатор Perl корректно завершает свою работу.

Упражнение по нахождению простых чисел

Какой же учебник по программированию обходится без подобного упражнения? В этом упражнении мы рассмотрим небольшую программу, которая находит и распечатывает простые числа. Как известно, простое число делится только на 1 и на само себя, например: число 2 — простое, 3 — простое, а 4 — составное (делится на 1, 4 и 2). Существует бесконечное множество простых чисел и их нахождение требует довольно большого количества вычислений.

Наберите в текстовом редакторе программу, приведенную в листинге 3.5, и сохраните ее под именем primes. **Не** нумеруйте **строки**. Сделайте файл выполняемым, следуя инструкциям, приведенным в конце 1-го занятия.

После этого попытайтесь запустить программу, набрав в командной строке:

```
perl primes
```

Листинг 3.5. Исходный код программы поиска простых чисел

```
#!/usr/bin/perl -w

$maxprimes=20; t Необходимо найти только первые 20 простых чисел
 $value=1;
  $count=0;
while($count < $maxprimes) {
     $value++;
     $composite=0;
OUTER: for($i=2; $i<$value; $i++) {
          for{$j=$i; $j<$value; $j++) {
             if (($j*$i)==$value) {
                 $composite=1;
                 last OUTER;
             }
          >
     }
   if (1 $composite) {
         $count++;
        print "Число $value простое\п";
     }
}
```

 Проведем анализ программы.

- *Строка 1.* В этой строке указан путь к интерпретатору (измените его в соответствии с конфигурацией вашей системы) и ключ `-w`. Всегда включайте режим выдачи предупреждений!
- *Строка 3.* Переменной `$maxprimes` присваивается максимальное количество целых чисел, которые нужно найти.
- *Строка 4.* Переменная `$value` будет содержать значение, проверяемое на принадлежность к простым числам.
- *Строка 5.* В переменной `$count` хранится счетчик найденных простых чисел.
- *Строка 6.* Цикл while выполняется до тех пор, пока не будет найдено достаточное количество простых чисел.
- *Строка 7.* Значение переменной `$value` увеличивается так, чтобы начать проверку на принадлежность к простым числам с числа 2.
- *Строка 8.* Переменная $composite используется в цикле for как флажок. Она обозначает, что на текущей итерации исследуемое число не является простым.
- *Строки 9—10.* Два вложенных цикла for перебирают все возможные множители числа `$value`. Например, для числа 4 будут проверяться пары 2 и 2, 2 и 3, 3 и 2, 3 и 3.
- *Строки 11-14.* Значения переменных $i и $j перемножаются. Если их произведение равно $value, для этой переменной устанавливается флаг $composite и программа выходит из обоих циклов.
- *Строки 17—20.* После циклов проверяется значение флага `$composite`. Если это Ложь, проверенное число является простым. В рассматриваемых строках выводится соответствующее сообщение и увеличивается значение счетчика.



Приведенный здесь алгоритм нахождения простых чисел не является самым быстрым или эффективным и служит лишь для демонстрации использования циклов. В любой книге, посвященной численным методам, вы сможете найти гораздо лучший алгоритм.

Резюме

В этой главе вы познакомились со многими управляющими конструкциями Perl. Некоторые из них, в частности оператор if и логические операторы, предназначены для выполнения определенных фрагментов кода в зависимости от истинности или ложности соответствующих значений. Другие операторы, такие как while, until и for, позволяют циклически выполнять фрагменты кода необходимое количество раз. Также вы узнали о логических значениях в Perl и их использовании во всех условных выражениях.

Вопросы и ответы

Мне приходилось программировать на С. Существуют ли в Perl аналоги операторов switch **и** case?

Нет! В Perl имеется такое огромное количество условных операторов, что даже трудно выбрать лучший способ эмуляции оператора switch. На мой взгляд, проще всего это сделать следующим образом:

```
if ($проверяемая_переменная == $звачевве1) {
    оператор1;
} elseif ($проверяемая_переменная == $звачевие2) {
    оператор2;
} else {
    оператор_по_умолчанию;
>
```

Страница руководства по синтаксису языка Perl, доступ к которой можно получить, набрав в командной строке perdoc perlsyn, содержит большое количество удачных примеров эмуляции оператора switch, некоторые из них имеют switch-подобный синтаксис.

Какое максимальное количество вложенных циклов for и while, а также операторов if допустимо?

Столько, сколько хотите, лишь бы хватило оперативной памяти. Обычно большое количество вложенных циклов означает, что вы выбрали неправильный подход к решению поставленной задачи.

Что мне делать? Perl выдает сообщение о том, что в программе отсутствует правая закрывающая фигурная скобка Unmatched right bracket (или Hissing right bracket). При этом номер строки с ошибкой соответствует концу файла.

Это означает, что в программе есть открывающая скобка { без парной ей закрывающей } или наоборот. Иногда Perl может угадать, где пропущена скобка, а иногда — нет. При глубоком вложении управляющих структур Perl не может найти ошибки, пока не будет проанализирован весь текст программы до конца файла. Хорошие программные редакторы (например vi, Emacs или MultiEdit) имеют средства, помогающие легко устранить несоответствие скобок. Воспользуйтесь ими.

Семинар

Контрольныевопросы

1. Оператор while выполняет цикл, пока условие истинно. Какой оператор выполняет цикл, пока условие ложно?

   а) if (not ...) {}

   б) while (! *условие) {}*

2. Истинно или ложно следующее выражение?

   `(O and 5) || (( "0" or 0 or "") and (6 and "Hello")) or 1`

а) истинно;

б) ложно.

3. Какое значение будет иметь переменная $i после окончания цикла?

```
for ($i=0; $i<=10; $i++) {    }
```

а) 10;

б) 9;

в) П.

Ответы

1. Правильным ответом будет вариант б). Цикл while (! *условие) {}* выполняется, пока условие ложно.

2. Правильный ответ — вариант а). Действия по упрощению этого выражения:

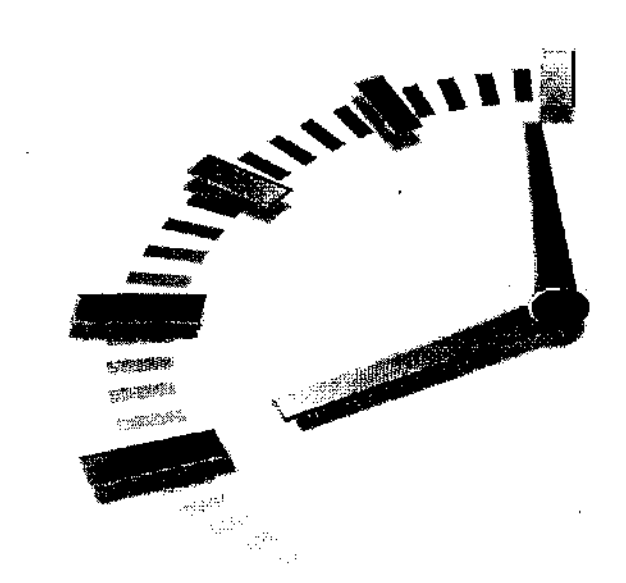
```
(Ложь) || ((Ложь) and (Истина)) or Истина
Ложь || Ложь or Истина
Истина
```

3. Правильным является вариант в). Проверочное условие `$i<=10`, поэтому, когда оно не выполняется, $i должно быть равно 11. Если вы ошиблись, не переживайте. Это довольно распространенная ошибка, имеющая у программистов специальное название, — *ошибка на единицу,* или *ошибка граничногоусловия.*

Упражнения

- Модифицируйте программу, приведенную в листинге 3.1, чтобы игра продолжалась, пока число не будет угадано.
- Программа, приведенная в листинге 3.5, неэффективна. Например, она анализирует четные числа больше 2, хотя очевидно, что они не могут быть простыми. Усовершенствуйте алгоритм поиска простых чисел.

4-й час



Укладка строительных блоков: списки и массивы

Скаляры Perl — существительные в единственном числе. Они могут представлять только один объект — слово, запись, документ, строку текста или символ. Однако иногда требуется оперировать целыми коллекциями объектов — набором слов, совокупностью записей, несколькими документами, 50 строками текста или десятками символов.

Множества объектов в Perl реализованы с помощью *списков данных.* Списки данных могут быть представлены тремя способами: с использованием простых списков, массивов и ассоциативных массивов.

Списки являются простейшей формой представления множества данных, по сути — это просто группа скаляров. Список представляет собой последовательность имен скаляров, разделенных запятыми. Вся последовательность заключена в круглые скобки. Например (2, 5, $a, "Bob") — список, состоящий из двух чисел, переменной $a и слова "Bob". Каждый отдельный скаляр называется *элементом списка.* Как и следовало ожидать, списки могут содержать произвольное количество элементов. Поскольку списки представляют собой коллекции скаляров, а скаляры также могут быть сколь угодно велики, то и списки могут быть довольно внушительных размеров.

Для хранения списка в виде переменной используется массив. Имена переменных-массивов в Perl подчиняются тем же правилам, что и имена остальных переменных, но начинаются с символа @. Например, @FOO — допустимое имя переменной-массива в Perl. Имена скаляров и массивов могут совпадать, например $names и @names — совершенно различные переменные. Первая обозначает скаляр, а вторая — массив. Они вообще могут не иметь никакого отношения друг к другу.

Индивидуальные скаляры, входящие в массив, называются *элементами массива.* На элементы массива можно ссылаться с помощью их положения в массиве — так называемого *индекса* (например, можно сослаться на третий элемент массива @FOO, пятый элемент массива @names и т. д.).

В Perl существует и еще один вид списков — *хэш,* или *ассоциативный массив,* который имеет много общего с обычным массивом. Подробнее об ассоциативных массивах речь пойдет на 7-м занятии, **"Хэши"**.

Основные темы этого занятия.

- Как заполнить пустой массив.
- Как поэлементно проверить массив.
- Как отсортировать и распечатать массив.
- Как преобразовать скаляры в массивы и как выполнить обратное преобразование.

Помещение скаляров в список или массив

Создать список литералов очень просто — достаточно заключить в скобки набор скалярных значений. Например:

```
(5, 'яблоко', $x, 3.14159)
```

В этом примере создается четырехэлементный список, содержащий число 5, слово 'яблоко', скалярную переменную $x и число я. Если список должен состоять только из обычных строк, можно обойтись без кавычек, для этого в Perl имеется оператор qw:

```
qw ( яблоки апельсины 45.6 $x )
```

Здесь создается четырехэлементный список. Каждый элемент может быть отделен от соседних символом пробела, табуляторами или символом перевода строки. Обратите внимание, что $x является литералом, т.е. строкой '$x', а не переменной $x, вместо которой подставляется ее значение. Если элементы списка должны содержать пробелы, то оператор qw использовать нельзя. Вместо него нужно использовать такую конструкцию:

```
('яблоки', 'апельсины', '45.6', '$x')
```

Обратите внимание, что литерал $x заключен в одинарные кавычки. Оператор qw не выполняет интерпретацию элементов списка, похожих на имена переменных, он воспринимает их буквально. Таким образом, '$x' не заменяется значением переменной $x, а оставляется в первоначальном виде.

В Perl имеется полезный оператор для работы со списками литералов — *оператор диапазона*. Оператор диапазона выглядит как пара точек (..). Ниже приведен пример использования этого оператора:

```
(1..10)
```

Оператор диапазона создает список, состоящий из всех целых чисел диапазона, заданного его левым и правым операндами (в нашем примере список будет состоять из целых чисел от 1 до 10 включительно). Если в списке должны быть представлены несколько диапазонов — используйте несколько операторов:

```
(1.-10, 20..30);
```

В этом примере создается список из 21 элемента: целые числа от 1 до 10 и от 20 до 30 включительно. Если правый операнд оператора диапазона меньше, чем левый, как в (10..1), создается пустой список.

Оператор диапазона можно применять не только к числам, но и к строкам. Диапазон (a..z) создает список, состоящий из 26 строчных букв. Диапазон (aa..zz) создает куда более длинный список, состоящий из 675 буквенных пар, начинающийся с aa, ab, ad и заканчивающийся zx, zy, zz.

Массивы

Списки литералов обычно используются для инициализации некоторых других структур, таких как массив или хэши. Для того чтобы создать в Perl массив, достаточно что-нибудь поместить в него. В отличие от других языков программирования, массив не нужно заранее объявлять и указывать его размерность. Создать новый массив и поместить в него элементы можно следующим образом:

```
@boys=qw { Гриша Петя Борис );
```

В этом примере инициализации массива используется оператор присваивания. Следует отметить, что оператор = применяется для присваивания значения как скалярам, так и массивам. После выполнения этого кода массив @boys будет содержать три элемента: Гриша, Петя и Борис. Обратите внимание на использование оператора qw, избавляющего от необходимости набирать шесть кавычек и две запятые. В присваивании значения элементам массива могут участвовать другие массивы и даже пустые списки:

```
@copy=@original;
@clean=();
```

В этом примере все элементы массива @original копируются в новый массив @copy. Если @copy содержал элементы, они будут потеряны. Второй оператор очищает массив @clean. Присваивание переменной пустого списка или пустого массива удаляет все ее элементы.

Если список литералов содержит другие списки, массивы или хэши, эти списки сводятся в один общий список, например:

```
@boys=qw ( Гриша Петя Борис );
@girls=qw( Наша Юля Света );
@kids=(@boys, @girls);
@family=(@kids, ('Миша', 'Катя'), 'Алиса');
```

Список (@boys, @girls) преобразуется Perl в один простой список имен всех детей (Гриша, Петя и т.д.) перед тем, как его значения присваиваются переменной @kids. В следующей строке кода массив Skids и список ('Миша', 'Катя') сводятся в один длинный список, затем этот список присваивается переменной @family. Первоначальные структуры @boys, @girls, @kids и ('Миша', 'Катя') не будут представлены в окончательном списке @family, там лишь находятся их элементы, такие как Миша и Катя. Это означает, что предыдущий пример создания массива @family эквивалентен такому оператору присваивания:

```
@family=qw(Гриша Петя Борис Наша Юля Света Миша Катя Аписа);
```

Если слева от оператора присваивания находится список имен переменных, эти переменные инициализируются элементами списка. Рассмотрим пример:

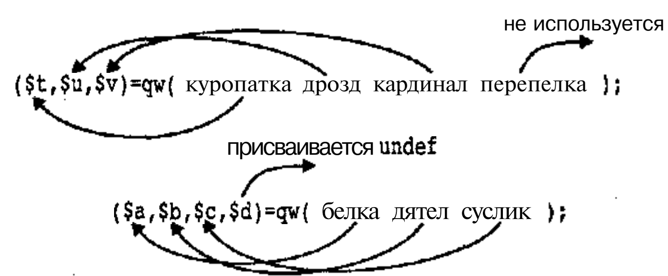
```
($a, $b, $c) = qw(яблоки апельсины бананы);
```

Здесь переменная $a инициализируется значением 'яблоки', $b — 'апельсины' и $c — 'бананы'. Если в списке слева содержится массив, этот массив "забирает" все возможные значения из правого списка. Например:



В этом примере переменной $a присваивается значение 'персики'. Остальные фрукты из правого списка присваиваются массиву @fruit. Переменной $c не достается значения, так как все значения "вобрал" в себя массив @fruit. Переменная $c становится неопределенной (undef).

Важно помнить, что если слева от оператора присваивания находится больше переменных, чем справа, избыточные переменные получают значение undef. Если справа больше элементов, чем слева, избыточные элементы справа просто игнорируются. Вот пример:



В первой строке переменным $t, $u и $v присваиваются значения из списка, находящегося в правой части оператора присваивания. Избыточный элемент правой части ('перепелка') попросту не используется в выражении. Во второй строке переменным $a , $Ь и $c присваиваются значения из списка, находящегося в правой части оператора присваивания, а переменной $d значения "не хватило" (последний элемент 'суслик' присваивается переменной $c). Следовательно, значение $d становится неопределенным (undef).

Доступ к элементам массива

Выше была описана методика работы с массивами и списками. Теперь рассмотрим вопрос доступа к их индивидуальным элементам. Такой доступ необходим для выполнения поиска элементов в массивах, изменения их значения, а также для добавления и удаления элементов.

Простейший способ получить доступ к содержимому всего массива — использовать его имя в двойных кавычках:

```
print "@array";
```

В этом примере будут распечатаны все элементы массива @array, разделенные пробелами.

Доступ к индивидуальным элементам массива осуществляется посредством их индексов. Индекс элементов массива начинается с 0 и с каждым элементом увеличивается на 1. Каждому элементу массива соответствует свое значение индекса, например:

@trees	0	1	2	3
	Дуб	Кедр	Клен	Ясень

Количество элементов массива ограничено лишь количеством доступной оперативной памяти. Для доступа к определенному элементу используется синтаксис *$имя_массива[индекс]*. Массив не обязательно должен существовать при обращении к его элементам. При необходимости массив создается автоматически. Ниже приведено несколько примеров работы с отдельными элементами массива:

```
@trees=qw( Дуб Кедр Клен Ясень );
print $trees[0];   f Печатает "Дуб"
print $trees[3];   # Печатает "Ясень".
$trees[4]='COCHa';
```

Как видите, когда речь идет об индивидуальных элементах массива @trees, в их названии используется символ $. Но ведь этот символ используется для обозначения скаляров, скажете вы. Символ $ в конструкции $trees[3] говорит о том, что это отдельный скаляр, находящийся в массиве @trees. Скаляры потому и обозначаются символом доллара, что содержат только одно значение. Это важный момент.

В начале этого занятия вы узнали, что скаляры и массивы могут иметь одинаковые имена, но при этом не быть связанными друг с другом. Perl усматривает разницу между скаляром Strees и элементом массива $trees[0], номер которого задается в квадратных скобках. Он понимает, что речь идет о первом элементе массива @trees, а вовсе не о скалярной переменной $trees.

Perl может оперировать с подгруппой внутри массива, называемой *сечением (slice)*. Сечение массива обозначается символом 6, свидетельствующим о том, что это группа элементов и квадратными скобками с перечислением индивидуальных элементов массива, например:

```
@trees=qw( дуб Кедр Клен Яблоня Вишня Сосна Персик Ель );
@trees[3,4,6];              # Фруктовые деревья
@conifers=@trees[5,7]       I Хвойные деревья
```

Определение размера массива

Часто требуется определить размер массива, или индекс его последнего элемента. Подобная задача возникает при добавлении или удалении элементов массива. В Perl для решения этой задачи предусмотрено несколько способов. Первый — использование специальной переменной $#*имя_массива*. Она возвращает последний допустимый индекс массива, например:

```
@trees=qw( Дуб Кедр Клен Ябловя Вишня Сосна Перснк Ель );
print $#trees;
```

В этом массиве восемь элементов, но, так как нумерация массивов начинается с нуля, печатается индекс 7. Изменение значения переменной $#trees изменяет длину массива. Уменьшение значения приводит к удалению элементов с большими индексами, а увеличение — добавляет в массив новые элементы. Новые элементы имеют неопределенное значение (undef).

Другой способ нахождения размера массива — использование имени массива в скалярном контексте (т.е. там, где в выражении ожидается скаляр), например:

```
$size=@array;
```

Переменная $size теперь содержит количество элементов массива @array. На этом примере мы продемонстрировали, как используется концепция контекста в Perl. О контексте мы поговорим в следующем разделе.



При работе с массивами можно также использовать отрицательные индексы, котрые соответствуют элементам, расположенным с конца массива. Например, $array[-1] — последний элемент массива @array, $array[-2] — следующий с конца и т.д.

Подробнее о контексте

Что же такое *контекст?* Контекст — это некое окружение элемента, помогающее понять, что он собой представляет. Если мы видим человека в одежде хирурга в больнице — скорее всего, он врач, а на бале-маскараде — один из гостей.

Люди используют Контекст для определения смысла слов. Например, слово *угол* может иметь несколько значений, в зависимости от окружающих его слов, или контекста:

- магазин за углом;
- прямой угол;
- снять угол.

Одно и то же слово, а значения разные. Значение слова зависит от окружающих его слов.

Точно так же и Perl реагирует на контекст. Функции и операторы Perl ведут себя по-разному в зависимости от контекста. Два наиболее важных контекста Perl — это *контекст списка* и *скалярный контекст.* Например, оператор присваивания (символ =) можно применять и к массивам, и к скалярам. Тип выражения, находящегося слева (список или скаляр), определяет контекст выражения, находящегося справа. Посмотрите на этот фрагмент кода:

```
$a=$b;       # Слева скаляр, следовательно, контекст скалярный.
@foo=@bar;   # Слева массив, контекст списка.
($a)=@foo;   t Слева список, опять контекст списка.
$b=@bar;     # Слева скаляр, скалярный контекст.
```

Последняя строка представляет особый интерес, потому что массивы в скалярном контексте возвращают количество элементов массива.

Сравните, как используются переменные $a и $b в следующих строках кода (обратите внимание, что оба оператора присваивания делают практически одно и то же):

```
@foo=qw( Вода Пепси Кола Лимонад );
$a=@foo;
$b=$|foo;
print "$a\n";
print "$b\n";
```

После выполнения этого кода переменная $a имеет значение 4, а $b — 3. Почему так происходит? Массив @foo в скалярном контексте возвращает количество своих элементов, которое присваивается переменной $a. А переменной $Ь присваивается значение индекса последнего элемента (не забывайте, что нумерация индекса массива начинается с нуля).

Учитывая, что массив в скалярном контексте возвращает количество своих элементов, легко проверить, пустой массив или нет:

```
@mydata=qw( Овес Пшеница Рожь Ячмень );
if (@mydata) {
    print "В массиве есть данные!\n";
}
```

Здесь массив @mydata используется в скалярном контексте, в результате выражение возвращает количество элементов массива, в нашем случае — 4. Условное выражение в операторе if равно 4, а значит, оно истинно и тело блока выполняется.



На самом деле массив @mydata здесь использован в специальном контексте, называемом *булевым,* или *логическим.* Это разновидность скалярного контекста со схожим действием. Булев контекст имеет место, когда Perl ожидает получить логическое значение, например в условном выражении оператора if. Еще одна разновидность контекста, называемая *пустым контекстом (void),* будет рассмотрена на 9-м занятии, "Дополнительные функции и операторы".

Возвращаясь к старой теме

Многие из операторов и функций Perl обусловливают контекст своих аргументов. Иногда действие операторов и функций также обусловлено контекстом. Некоторые из этих функций нам уже встречались, но мы не обращали внимание на такие их свойства.

Функция print ожидает в качестве аргумента список. Неважно, в каком контексте формируется этот список. Поэтому функция print с массивом @foo, переданным в качестве аргумента, распечатывает элементы этого массива, находящегося в контексте списка:

```
print @foo;
```

Для навязывания скалярного контекста можно использовать псевдофункцию scalar:

```
print scalar(@foo);
```

Здесь печатается количество элементов массива @foo. Функция scalar определяет скалярный контекст для ifoo, поэтому массив возвращает количество своих элементов, выводимое впоследствии функцией print.

Рассмотренной нами на 2-м занятии, "Строительные блоки Perl: числа и строки", функции chomp может быть передан в качестве аргумента как скаляр, так и массив. В скалярном аргументе удаляется завершающий символ-разделитель записей. Если аргументом является массив, символ-разделитель удаляется в конце каждого скалярного элемента.

Вы уже знаете, как прочитать данные, вводимые с клавиатуры, используя **оператор** <STDIN>. Угловые скобки — это специальный оператор Perl, который, в зависимости от контекста, ведет себя по-разному. В скалярном контексте этот оператор читает с терминала одну строку. В контексте списка этот оператор читает весь поток данных, поступающих с терминала, до символа конца файла и помешает затем все данные в список. Сравните:

```
$a=<STDIN>;        # Скалярный контекст,
                   # в переменную $a помещается одна строка.
@whole=<STDIN>;    # Контекст списка, все введенные строки
                   # помещаются в массив @whole.
($a)=<STDIN>;      # Контекст списка, все введенные строки
                   # помещаются в список.
```

Какое значение получает переменная $a в третьем примере? Помните, выше мы говорили, что если в левой части оператора присваивания находится список, причем количество его элементов меньше, чем в списке, находящемся в правой части, то избыточные элементы правой части отбрасываются. Таким образом, в третьем примере будут считаны все данные с терминала, но в переменную $a помещается только первая введенная строка.



> Что такое конец *файла?* После окончания ввода с терминала надо дать знать Perl, что ввод данных завершен. Для этого нужно набрать символ конца файла (EOF). Что это за символ, зависит от операционной системы. В UNIX таким символом является <Ctrl+D>, помещенный в начале строки. В MS-DOS или Windows признаком конца файла являются два идущих подряд символа <Ctrl+Z>, которые могут располагаться в любом месте текста.

Оператор повторения, рассмотренный на 1-м занятии, "Начало работы с Perl", в контексте списка ведет себя специфическим образом. Если левый операнд оператора повторения взят в скобки и сам оператор находится в контексте списка, то возвращается список с элементами, соответствующими левому операнду. В этом примере создается список из 100 звездочек:

```
@stars= C*") x 100;
```

Левый операнд "*" оператора повторения находится в скобках, а значение полученного выражения присваивается массиву, что и определяет контекст списка. Такой синтаксис применяется для инициализации элементов массива одинаковыми значениями.

Другой часто используемый оператор, который вообще мало кто относит к категории операторов, — запятая (,). Пока мы лишь использовали ее для разделения элементов списка литералов, например:

```
@pets=('Кошка', 'Собака', 'Рыбки', 'Канарейка', 'Игуана');
```

Поскольку здесь запятая находится в контексте списка, она выполняет свои обычные функции — разделение элементов списка. Однако запятая в скалярном контексте ведет себя иначе. Этот оператор вычисляет значение каждого элемента списка слева направо и возвращает значение крайнего правого элемента, например:

```
$last_pet=('Кошка', 'Собака', 'Рыбки', 'Канарейка', 'Игуана');
f Совсем не то, что вы могли подумать!
```

Здесь названия домашних животных, расположенные справа от оператора присваивания, с точки зрения Perl не являются списком. Это группа строковых литералов, значение которых вычисляется в скалярном контексте слева направо (из-за скаляра $last_pet, расположенного в левой части). В результате переменной $last_pet присваивается значение 'Игуана'.

Другой пример — функция localtime, в зависимости от контекста, имеет два абсолютно различных варианта поведения. В скалярном контексте функция localtime возвращает форматированную строку текущего времени. Например, оператор print scalar(localtime) напечатает что-то похожее на Thu Apr 13 10:14:45 2000. В контексте списка функция localtime возвращает список элементов, описывающих текущее время:

```
($sec, $min, $hour, $mday, $mon, $year_off, $wday, $yday, $isdst)=localtime;
```

Значения этих элементов приведены в табл. 4.1.

Таблица 4.1. Возвращаемые значения функции localtime в контексте списка

Поле	Значение
$sec	Секунды, 0-59
$min	Минуты, 0-59
$hour	Часы, 0-23
$mday	День месяца, 1-28, 29, 30 или 31

Окончание табл. 4.1

Поле	Значение
$mday	День месяца, 1-28, 29, 30 или 31
$mon	Месяц, 0-.11 (обратите внимание!)
$year_off	Количество лет, прошедших с 1900 года. Прибавив к этому числу 1900, вы получите корректное значение текущего года
$wday	День недели, 0-6
$yday	День года, 0-364 или 365
$isdst	Истина, если действует летнее время



Большинство проблем 2000 года в программах на Perl были связаны с неправильным использованием параметра `$year_off`, возвращаемого функцией `localtime`. Чтобы получить текущий год, большинство программистов добавляло к значению этого параметра строку '19'. Однако следует учитывать тот факт, что разница между текущим и 1900 годом в 1999 году равна 99, в 2000 году — 100. Арифметическое же сложение этого значения с 1900 будет корректно работать и после 2000 года. Сам Perl давно избавлен от ошибки Y2K, но использование параметра `$year_off` с префиксами '19' или '20' действительно может привнести ее в вашу программу.

Как же узнать, какой контекст создают аргументы определенной функции или оператора и как эта функция или оператор ведет себя в различных контекстах? Это не так-то просто, поскольку единой методики нет. В документации представлены все функции и операторы с объяснением их поведения в зависимости от нескольких факторов. В дальнейшем мы будем обращать ваше внимание, если встретившаяся впервые в этой книге функция специфическим образом ведет себя в различных контекстах.

Работа с массивами

Теперь, после знакомства с основными правилами построения массивов, пришло время изучить несколько полезных методов работы с ними.

Поэлементная работа с массивом

На 3-м занятии, "Управление процессом выполнения программы", были рассмотрены циклы while, for и некоторые другие конструкции управления течением программы. Многие задачи, выполнение которых требует использования массивов, связаны с поэлементной обработкой массива, называемой *итерацией*. Один из способов организации итерации — использование цикла for, например:

```
@flavors=qw( Шоколадное Ванильное Клубничное Пломбир Фруктовое};
for ($index=0; $index<@flavors; $index++) {
    print "Мой любимый сорт мороженого — $flavors($index)";
}
print ", а также все остальные.\п";
```

В первой строке инициализируется массив названий различных сортов мороженого. Для простоты кода использован оператор qw. Если бы в списке были названия, состоящие из нескольких слов, например Крем-брюле, пришлось бы использовать син-

таксис с одинарными кавычками. Во второй строке выполняется основная часть работы- Счетчик $index инициализируется значением 0 и циклически увеличивается на единицу, пока не будет достигнуто значение §flavors. В данном случае @flavors находится в скалярном контексте и имеет значение 5 — количество элементов массива.

Не правда ли, что для такой простой задачи, как перебор элементов массива, нужно выполнить большое количество работы? Наверняка в Perl должны быть предусмотрены средства, позволяющие упростить громоздкий код. Данный случай — не исключение. В Perl имеется, еще не рассмотренный нами, оператор цикла foreach. Оператор foreach устанавливает индексную переменную, называемую *итератором,* принимающую в цикле значение каждого элемента списка. Рассмотрим следующий пример:

```
foreach Scone (@flavors) {
    print "Я люблю рожок мороженого из $cone\n";
}
```

Здесь переменная $cone последовательно принимает значение каждого элемента массива @flavors. Как только $cone принимает значение очередного элемента массива @flavors, выполняется тело цикла, выводящее сообщение с этим элементом. Не забывайте, что в цикле foreach итератор не просто принимает значения всех элементов списка. Фактически итератор здесь используется как указатель, поэтому модификация переменной $cone в теле цикла приводит к модификации соответствующих элементов массива @flavors. Вот пример:

```
foreach $flavor (@flavors) {
  $flavor="$flavor мороженое"; # Происходит модификация массива @flavors!
    Print "Я люблю $flavor";
}
```

Во второй строке происходит модификация переменной $flavor — к ее значению добавляется слово мороженое. В третьей строке выводится фраза "Я люблю шоколадное мороженое", а затем подобные строки печатаются и для остальных сортов мороженого. После окончания цикла окажется, что к каждому элементу массива @flavors добавлено слово мороженое.



В Perl служебные слова foreach и for — синонимы. Любое из них можно использовать вместо другого. В этой книге для ясности оператор foreach() используется для итерации в массиве, а оператор for() — для обычных циклов, о которых шла речь на 3-м занятии, "Управление процессом выполнения программы". Имейте в виду, что эти операторы взаимозаменяемы.

Взаимные преобразования массивов и скаляров

В Perl нет общего правила для преобразования массивов в скаляры и наоборот. Вместо этого предлагается набор функций и операторов для выполнения таких преобразований. Один из способов преобразования скаляра в массив предполагает использование функции split. Этой функции передается в качестве второго аргумента скаляр, который она разбивает на элементы массива в соответствии с шаблоном, заданным первым аргументом (в примере он заключен в символы косой черты).

```
@words=split(/ /, "В лесу родилась елочка");
```

После выполнения этого кода массив @words будет содержать слова В, лесу, родилась и елочка без пробелов. Если исходная строка не определена, используется значение переменной $_. Если функция split вызывается без параметров, то выполняется разбиение на слова переменной $_. При этом в качестве символа-разделителя используется пробел. Существует также специальный шаблон разбиения '' (нулевой шаблон), разделяющий скаляр на индивидуальные символы, как показано ниже:

```
while (<STDIN>) {
      ($firstchar)=split(//, $_);
      print "Первый символ введенной строки — $firstchar\n";
>
```

В данном примере с терминала считывается отдельная строка и помещается в переменную $_. Следующая строка кода разбивает переменную $_ на отдельные символы. При этом используется нулевой шаблон. В результате функция split возвращает список всех символов, находящихся в переменной $_. Этот список присваивается списку, расположенному слева от оператора присваивания. Первый элемент списка присваивается переменной $firstchar, а остальные элементы отбрасываются.



Используемые в операторе split шаблоны называются *регулярными выражениями.* Регулярные выражения — это довольно сложная тема, и мы вернемся к ней на 6-м занятии, "Поиск по шаблону". А пока для примеров мы будем использовать простые шаблоны, такие как пробелы, двоеточия и т.п. После того как вы познакомитесь с регулярными выражениями, я приведу примеры использования более сложных шаблонов для разделения скаляров с помощью оператора split.

Описанный метод преобразования скаляра в список часто используется в Perl. При разделении скаляра, имеющего определенную структуру, например записи с полями, удобно использовать для частей имена, например:

```
@Music=('White Album,Beatles',
        'Graceland,Paul Simon',
        'A Boy Named Sue, Goo Goo Dolls');
foreach $record (@Music) {
     ($record_name, $artist)=split{/,/, $record);
}
```

При использовании списка переменных сразу аидно, что первое поле записи — это название альбома, а второе поле — исполнитель. При обычном преобразовании в массив значения полей не были бы столь очевидными.

Кроме разбиения, существует и обратная операция — слияние массивов и образование скаляров. Для этой цели в Perl предусмотрена функция join. Ей передается строка-разделитель и список элементов, которые нужно объединить. Вот пример:

```
$number=join(',', (1..10));
```

Здесь переменной $number присваивается строка 1,2,3,4,5,6,7,8,9,10. Вы можете использовать функции split и join для разделения и объединения строк, причем возвращаемое значение одной функции может поступать на вход другой:

```
$message="TyT был Вася";
print "Строка \"$message\" состоит из:",
        join('-', split(//, $message));
```

В этом примере переменная $message преобразуется в список с помощью функции split. Этот список поступает на вход функции join и заново преобразуется в строку, но уже с дефисами. В результате выполнения этого кода будет выведено следующее сообщение:

Строка "Тут был Вася" состоит из:Т-у-т- -б-ы-л- -В-а-с-я

Упорядочивание элементов массива

Иногда требуется изменить порядок следования элементов в массиве. Например, программа на Perl должна прочитать из файла список пользователей, отсортировать их по алфавиту и распечатать. Для сортировки в Perl предусмотрена функция sort, которой в качестве аргумента передается список. Функция возвращает другой список, отсортированный по алфавиту. Исходный массив при этом не модифицируется. Вот пример:

```
@Chiefs=qw(Клинтон Буи Рейган Картер Форд Никсон);
print join(' ', sort @Chiefs);
```

После выполнения этого кода будет выведена строка: Буш Картер Клинтон Никсон Рейган Форд. Имейте в виду, что установленный по умолчанию порядок сортировки использует значения кодов ASCII. Это означает, что символы верхнего регистра имеют преимущество перед символами нижнего регистра. Числа при этом сортируются совсем не так, как вы могли бы ожидать. Они сортируются не по значению. Например, 11 идет после 100. Для сортировки по значению необходимо использовать порядок, отличный от заданного по умолчанию.

Функция sort позволяет выполнять сортировку в нужном вам порядке. Для этого ей в качестве первого параметра необходимо передать код или имя подпрограммы. Внутри блока или подпрограммы используются две переменные $a и $b, которые соответствуют двум элементам списка. Задача блока возвратить -1, 0 или 1, если $a меньше $Ь, $a равно $b или $a больше $b соответственно. Ниже приведен один из способов сортировки чисел. В массиве @numbers содержатся числовые значения, которые нужно отсортировать.

```
@sorted=sort { return(1) if ($a>$b);
               return (0) if ($a==$b);
               return(-1) if ($a<$b);} @numbers;
```

Этот пример, конечно же, будет сортировать числа, но его код выглядит слишком сложно для такой простой задачи. Как и следовало ожидать, в Perl есть замена такой сложной конструкции — специальный оператор, который в шутку называют "космическим кораблем" <=>. Этот оператор получил свое название из-за того, что он напоминает летающую тарелку (вид сбоку). Он возвращает -1, если левый операнд меньше правого, 0 — если операнды равны, 1 — если левый операнд больше правого:

```
@sorted=sort { $a<=>$b;> @numbers;
```

Этот код компактнее, проще и легче читается. Оператор "космический корабль" можно использовать лишь для сравнения числовых значений.

Для сравнения строк используется оператор cmp, работающий подобным образом. Вы можете использовать гораздо более изощренный критерий сортировки, реализовав его в виде сложной подпрограммы сортировки. Примеры такой сортировки можно посмотреть в разделе 4 Perl FAQ.

И последняя функция, которую мы рассмотрим на этом занятии, — reverse. Она очень простая. В скалярном контексте в качестве параметра ей передается скаляр. Функция обращает порядок следования символов и возвращает полученную строку. Например, reverse("Perl")в скалярном контексте возвращает lreP. В контексте списка функция reverse возвращает список, элементы которого расположены в обратном порядке:

```
@lines=qw(Не будем прогибаться под изменчивый мир);
print join(' ', reverse sort @lines);
```

В этом примере вначале выполняется функция sort, которая сортирует список (Не, будем, прогибаться, под, изменчивый, мир). Этот список перестраивается в обратном порядке с помощью функции reverse и передается в качестве параметра функции join для объединения в строку. В качестве разделителей используются пробелы. Результат — прогибаться под мир изменчивый будем Не. Правда, трудно не согласиться с этим утверждением?

Упражнение: небольшая игра

На этом занятии вы получили большое количество новой, зачастую неожиданной, информации. Оказалось, что знакомые операторы ведут себя совершенно по-разному в зависимости от контекста. Вы познакомились с новыми операторами и функциями, а также изучили несколько новых синтаксических правил. Чтобы вы не впали в депрессию от такого обилия информации, я придумал игру, способную нейтрализовать ваши негативные эмоции.

Наберите в текстовом редакторе программу, приведенную в листинге 4.1, и сохраните ее в файле Hangman. Сделайте файл выполняемым, согласно инструкциям, приведенным на 1-м занятии.

После этого попытайтесь запустить программу, набрав в командной строке

```
perl Hangman
```

В листинге 4.2 приведен пример диалога с программой Hangman.

Листинг 4.1. Исходный текст программы Hangman

```
#!/usr/bin/perl -w

@words=qw( Интернет Ответ Принтер Программа );
$guesses[0]="";
$wrong=0;

$choice=$words[rand @words];
$hangman="0-|--<";

@letters=split(//, $choice);
@hangman=split(//, $hangman);
@blankword=(0) x scalar(@hangman);
OUTER:
    while ($wrong<@hangman) {
        foreach $i (0..$#letters) {
                if ($blankword[$i]) {
                        print $blankword[$i];
                } else {
                        print "-";
                }
        }
```

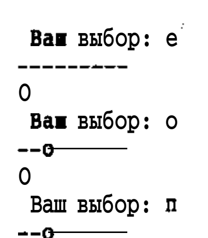
```
22:       print "\n";
23:       if ($wrong) {
24:              print @hangman[0..$wrong-1]
25:        }
26:      print "\n Ваш выбор: ";
27:       $guess=<STDIN>;  chomp $guess;
28:       foreach(@guesses) {
29:            next OUTER if ($_ eq $guess);
30:        }
31:        $guesses[$#guesses]=$guess;
32:        $right=0;
33:      for ($i=0; $i<@letters; $i++) {
34:            if ($letters[$i] eq $guess) {
35:                     $blankword[$i]=$guess;
36:                     $right=1;
37:              }
38:        }
39:       $wrong++ unless($right);
40:      if (join('', @blankword) eq $choice) {
41:             print "Вы угадали!\n";
42:              exit;
43:        }
44: }
45: print "$hangman\n Печально, но было загадано слово $choice.\n";
```

Проведем анализ протраммы.

- *Строка 1.* В этой строке находится путь к интерпретатору (измените его в соответствии с конфигурацией вашей системы) и ключ -w. Всегда включайте режим выдачи предупреждений!
- *Строка 3.* Массив Swords инициализируется списком допустимых в этой игре **слов.**
- *Строки 4-5.* Инициализируются некоторые переменные. Массив `@guesses` служит для хранения ранее введенных букв. Переменная $wrong содержит количество неудачных ответов.
- *Строка 7.* Из массива `@words` случайным образом выбирается слово и присваивается переменной $choice. Функции rand{) должен передаваться скалярный аргумент, поэтому конструкция Swords воспринимается как скаляр. Значение Swords в скалярном контексте — количество элементов массива Swords, в данном случае 4. Функция rand возвращает случайное число в диапазоне от 0 до 4, не включая крайние значения. При использовании числа с плавающей точкой в качестве индекса массива знаки после запятой отбрасываются.
- *Строка 8.* Формируется фигурка.
- *Строка 10.* Загаданное слово разбивается на буквы, которые помещаются в массив `@letters`.
- *Строка 11.* Фигурка разбивается на части, которые помещаются в массив `@hangman`. Причем `$hangman[0]` — голова, `$hangman[1]` — шея и т.д.

- *Строка 12.* Массив @blankword предназначен для отображения положения правильно угаданных букв. Вначале в @blankword находится список (0) x scalar(@hangman), длина которого равна количеству элементов @hangman. Затем постепенно нули заменяются на угаданные буквы. Это делает строка 35 кода.
- *Строки 13–14.* Это основной цикл программы. У него есть метка OUTER", позволяющая использовать специальные операторы управления циклом. Этот цикл выполняется, пока количество неправильных ответов не сравняется с длиной фигурки.
- *Строки 15–21.* Цикл foreach проверяет элементы массива @blankword, и все угаданные буквы распечатываются, а еще не угаданные заменяются дефисами.
- *Строки 23–25.* Переменная $wrong содержит количество неправильных ответов. Если имеется хотя бы один неправильный ответ, печатаются $wrong начальных элементов массива @ hangman.
- *Строки 26–27.* Эти строки вводят ответ игрока. Функция chomp удаляет завершающий символ новой строки.
- *Строки 28-30.* Эти строки проверяют, не вводился ли символ ранее. Если да, цикл снова начинается со строки 13. Игрок не наказывается за повторение неправильного ответа.
- *Строка 31.* Введенная пользователем буква помешается в массив @guesses.
- *Строки 32–38.* Это сердцевина программы. В массиве @letters, содержащем загаданное слово, ищется буква ответа. Если буква найдена, она присваивается соответствующему элементу массива @blankwords. Все элементы массива @blankwords — это или угаданные буквы, или нули. Флаг $right получает значение 1, если хотя бы одна буква правильно угадана.
- *Строка 39.* Переменная $wrong увеличивается на единицу при каждом неправильном ответе пользователя.
- *Строки 40–43.* Элементы массива @blankwords объединяются в строку, которая сравнивается с исходным словом. Если они совпадут — это означает, что пользователь полностью угадал слово.
- *Строка 45.* Основной цикл программы завершается, поскольку **игрок** исчерпал все свои попытки. Программа выводит сочувственное сообщение и выходит **из игры.**

Листинг 4.2. Образец диалога с программой Hangman



```
---------

 Ваш выбор: е
---------
0
 Ваш выбор: о
--о------
0
 Ваш выбор: п
--о------
```

```
0-
 Ваш выбор: П
П-о......
0-
 Ваш выбор: р
Про-р——
0-
 Ваш выбор: г
Прогр——
0-
 Ваш выбор: а
```

В этой небольшой программе я постарался продемонстрировать весь изученный в этом часе материал — списки литералов, массивы, функции split и join, контекст и циклы foreach. Подобную игру можно было бы запрограммировать массой различных способов, наша же программа предназначена для иллюстрации основных возможностей массивов.

Резюме

Массивы и списки предназначены для хранения набора объектов Perl. В них может находиться практически неограниченное количество скаляров. С ними можно обращаться как с единым целым и оперировать их отдельными элементами. Perl позволяет легко копировать массивы, сортировать их, объединять несколько массивов и преобразовывать скаляры в массивы и наоборот. Работа большинства операторов и функций Perl зависит от контекста. Они по-разному ведут себя в скалярном контексте и контексте списка.

Вопросы и ответы

Существует ли быстрый способ поиска определенной строки в элементах массива?

Обычно используют итерацию в массиве с проверкой каждого его элемента. Если такую проверку необходимо делать много раз, для поиска элементов лучше воспользоваться возможностями ассоциативных массивов или хэшей, рассматриваемых на 7-м занятии.

Как удалить повторяющиеся элементы массива?

Как подсчитать количество уникальных элементов?

Как проверить, содержатся ли в двух различных массивах одинаковые элементы?

Ответ тот же: используйте хэш. Хэши позволяют быстро и эффективно выполнять различные манипуляции над массивами. Все эти вопросы будут рассмотрены на 7-м занятии.

Семинар

Контрольные вопросы

1. Какой наиболее эффективный способ поменять значения переменных $a и $b.

   а) `$a=$b;`

   б) `($a,$b)=($b,$a);`

   в) `$c=$a; $a=$b; $b=$c;`

2. Какое значение получит переменная $a после выполнения оператора `$a=scalar(@array);` ?

   а) количество элементов массива `@array`;

   б) индекс последнего элемента массива `@array`;

   в) такой синтаксис недопустим.

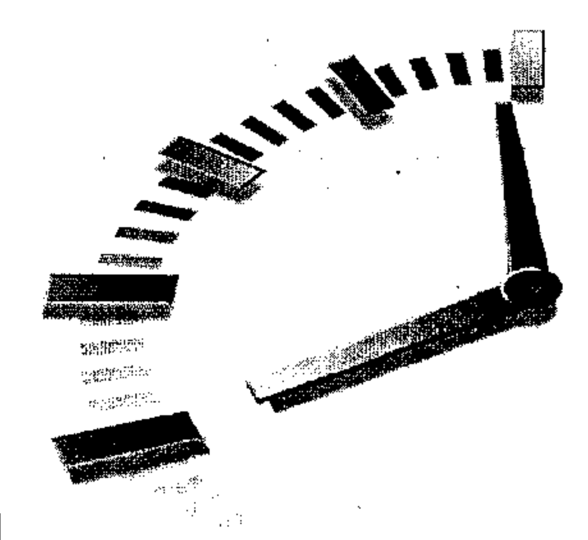
Ответы

1. Правильным является вариант б). Первый пример вообще не будет работать, поскольку начальное содержимое переменной $a будет потеряно. Вариант в) подходит, но в нем используется вспомогательная переменная. Простой код варианта б) корректно меняет значения переменных и не требует введения дополнительных переменных.

2. Правильный ответ — вариант а). Массив в скалярном контексте возвращает количество своих элементов. Для определения индекса последнего элемента используется конструкция `$#array`. Использование функции scalar в этом примере не обязательно, поскольку в левой части оператора присваивания находится скаляр. Он и определяет скалярный контекст для массива `@array`.

Упражнения

- Модифицируйте программу Hangman так, чтобы фигурка печаталась в вертикальном положении.



5-й час

Работа с файлами

До сих пор наши программы работали автономно. Единственно доступными для них средствами связи с внешним миром были вывод сообщений, предназначенных пользователю, и ввод данных с клавиатуры. Однако отныне все изменится!

Perl — язык с исключительными возможностями работы с *файловым вводом-выводом.* Скаляры Perl могут вместить запись любой возможной длины, а массивы — содержимое целых файлов, если, конечно, позволяет оперативная память компьютера. С данными, содержащимися в скалярах и массивах Perl, можно производить многочисленные манипуляции и записывать их в новые файлы.

Perl создавался с учетом максимального облегчения выполнения операций файлового ввода-вывода. Некоторые его встроенные операторы оптимизированы для выполнения типичных операций ввода-вывода. На этом занятии вы узнаете, как получить доступ к данным в файлах.

Основные темы этого занятия.

- Как открывать и закрывать файлы.
- Как записывать данные в файлы.
- Как читать данные из файлов.
- Как создавать "безопасные" программы.

Открытие файлов

ДЛЯ чтения и записи файлов в Perl необходимо открыть так называемый *дескриптор файла.* Дескрипторы файлов — еще одна разновидность переменных Perl. Они служат для идентификации файлов в программе и операционной системе. В дескрипторе содержится информация о способе открытия файла, режимах доступа (чтение и/или запись), а также атрибуты, определенные пользователем.

Из материала прошлых занятий вы уже знакомы с одним из дескрипторов — STDIN. Этот дескриптор автоматически передается программе при запуске и обычно связан с клавиатурой (позднее вы узнаете еще некоторые особенности дескриптора STDIN).

Формат имен дескрипторов тот же, что и имен других переменных Perl. Подробнее об этом шла речь на 2-м занятии, "Строительные блоки Perl: числа и строки". Единственное отличие — в именах дескрипторов файлов не должно быть символов, идентифицирующих тип переменной ($, @ или какого-нибудь другого). Поэтому рекомендуется в именах дескрипторов использовать только символы верхнего регистра, чтобы они случайно не совпали с современными или будущими зарезервированными служебными словами Perl, такими как foreach, else, if и т.д.



> Для имени дескриптора можно использовать строковой скаляр или функцию, возвращающую строку. Такие дескрипторы файлов называются *косвенными.* Сведения об их использовании обычно не приводятся в учебных пособиях для начинающих. За дополнительной информацией о косвенных дескрипторах файлов обращайтесь к документации: страница perlfunc, раздел open.

Каждый раз, когда необходимо получить доступ к файлу на диске, необходимо создать новый дескриптор и открыть его. Для открытия дескрипторов, как вы уже наверное догадались, используется функция open:

open (*дескриптор_файла*, *путь*)

Первый аргумент функции open — дескриптор файла, второй аргумент — путь. Путь указывает, какой файл необходимо открыть, поэтому, если не указан полный путь, например c:/windows/system/, функция open попытается открыть файл в текущем каталоге. При успешном выполнении функция open возвращает ненулевое значение (Истина), при неудачном — возвращается undef (Ложь), например:

```
if (open(MYFILE, "mydatafile")) {
      # Выполняется при успешном открытии
} else {
      print "Ошибка при открытии файла mydatafile!\n";
      exit 1;
}
```

Здесь при удачном завершении функция open возвращает истинное значение, открывается дескриптор файла MYFILE и выполняется блок if. В противном случае файл не открывается, выполняется блок кода else, сообщающий об ошибке. Во многих программах Perl подобный синтаксис "открыть или сообщить об ошибке" может быть реализован с помощью функции die. Функция die останавливает выполнение программы и выводит сообщение об ошибке:

Died at *имя_сценария* line *xxx*

Здесь *имя_сценария* — название программы на Perl, *xxx* — номер строки, в которой встретилась функция die. Функции die и open часто используются вместе следующим образом:

```
open(MYTEXT, "novel.txt") || die;
```

Программа или открывает файл, или прекращает свое выполнение. Если open завершилась неудачно, возвратив ложное значение, вступает в действие логический оператор ИЛИ (||). В результате будет вычисляться аргумент, находящийся в правой части оператора (в данном случае — функция die). При удачном выполнении функции open правая часть логического выражения не вычисляется. Иногда используют другой вид логического ИЛИ — or.

По окончании работы с дескриптором его следует закрыть. Это хороший стиль программирования. Операция закрытия сообщает операционной системе, что указанный дескриптор следует освободить для повторного использования, а находящиеся в памяти данные, связанные с файлом, — записать в файл (если они не были сохранены ранее). Следует отметить, что операционная система позволяет открыть ограниченное количество дескрипторов файлов. После достижения этого предела для открытия нового дескриптора придется закрыть один из уже открытых. Для закрытия дескриптора используется функция close:

```
close(MYTEXT);
```

Если попытаться открыть файл, указав в качестве параметров функции open один из уже открытых дескрипторов, то вначале этот дескриптор закрывается, а затем повторно открывается.

Пути

До сих пор мы открывали файлы, указывая только их имена, например novel.txt. При открытии файлов без указания имени каталога Perl считает, что файл находится в текущем каталоге. Чтобы открыть файл в другом каталоге, необходимо указать *путь*. Путь указывает операционной системе, где находится файл.

Путь нужно указывать в виде, принятом в используемой операционной системе. Ниже приведено несколько примеров путей для различных операционных систем:

```
open(MYFILE, "DISK5:[USER.PIERCE.NOVEL]") || die;    # VMS
open(MYFILE, "Drive:folder:name") || die;            # Macintosh
open(MYFILE, "/usr/pierce/novel") || die;            # UNIX
```

В системах Windows и MS-DOS в качестве разделителей в путях можно использовать символы обратной косой черты, например \Windows\users\pierse\novel.txt. Только при этом нужно помнить, что в строках, заключенных в двойные кавычки, символ обратной косой черты означает, что следующий за ним символ является специальным. Например:

```
open (MYFILE, "\Windows\users\pierse\novel.txt ") || die; # Ошибка!
```

Этот пример не сработает, потому что \n в строках в двойных кавычках обозначает символ перевода строки, а не букву n, а все остальные обратные косые черты будут просто игнорироваться Perl. Вот корректный способ открыть файл:

```
open (MYFILE, "\\Windows\\users\\pierse\\novel.txt ") || die;
# Правильно, но некрасиво
```

Чтобы эта же строка выглядела красивее, используйте символ косой черты (/) как разделитель путей в Windows и MS-DOS (в Windows и MS-DOS это допускается):

```
open (MYFILE, "/Windows/users/pierse/novel.txt ") || die;
# Значительно лучше
```

Пути могут быть как абсолютными (/home/foo в UNIX или c:/windows/win.ini в Windows), так и относительными (../junkfile в UNIX или ../bobdir/bobsfile.txt в Windows). Функция open в Windows способна воспринимать пути, следующие соглашению об универсальных именах UNC (Universal Naming Convention). Формат путей в UNC выглядит так:

```
\\имя_машины\имя_ресурса
```

Perl понимает пути, заданные в формате UNC с использованием как прямых, так и обратных косых черточек, открывает файлы на удаленных системах, если сетевые средства операционной системы позволяют это сделать, например:

```
open (REMOTE, "//fileserver/common/foofile") || die;
```

В Macintosh путь состоит из имени тома, за которым следуют имена папки и файла, разделенные символами двоеточия, как показано в табл. 5.1.

Таблица 5.1. Спецификаторы пути в MacPerl

Путь в Macintosh	Описание
`System:Utils:config`	Системный диск, папка `Utils`, файл config
`MyStuff:friends`	Папка `MyStuff` в текущей папке, файл friends
`ShoppingList`	Текущий диск, текущая папка, файл ShoppingList

Береженого Бог бережет

Создание программ для компьютера часто сопровождается неоправданным оптимизмом у программистов. Они думают: "Вот теперь она работает как надо!" или "Все ошибки наконец-то исправлены". Вообще чувство гордости за проделанную работу — отличная вещь; все достижения принадлежат людям, которые пытаются сделать невозможное возможным. Но иногда самоуважение переходит все границы и превращается в самоуверенность или высокомерие. В одной из древнегреческих трагедий высокомерие всегда жестоко наказывалось богами.



Описанный феномен стал проявляться по мере распространения компьютеров. Фредерик П. Брукс (Frederic P. Brooks) в своей классической работе *The Mythical Man-Month* (Addison Wesley, 1975, с. 14) писал: "Все программисты — неисправимые оптимисты. Возможно, современное волшебство (программирование) особенно привлекает тех, кто верит, что все сказки имеют хороший конец, но... Все наши идеи ложны, нам свойственно ошибаться, поэтому наш оптимизм неоправдан".

До настоящего времени все приведенные примеры и упражнения имели дело с внутренними данными программы (множители чисел, массивы данных и т.д.) или с простыми строками, введенными пользователем с клавиатуры. При работе с файлами программы часто имеют дело с неподконтрольными им данными. Особенно это актуально, если данные не расположены на вашем компьютере, а пересылаются по сети. Поэтому следует иметь в виду, что может произойти сбой, на который программа должна отреагировать должным образом. Написание подобных программ называется *безопасным программированием.* "Безопасная" программа имеет бесспорные преимущества перед обычной, особенно если это касается устойчивости работы.

Если программа взаимодействует с внешним миром, например открывает дескриптор файла, прежде чем продолжить выполнение, *всегда* нужно удостовериться, что предыдущая операция выполнена успешно. Лично я отлаживал сотни программ, в которых программист использует вызовы операционной системы, без проверки результата их работы, что очень часто приводит к сбою. Даже если ваша программа тестовая и пишете вы ее на скорую руку, всегда полезно убедиться, что все произошло так, как и ожидалось.

Умирать, так с музыкой

Функция die используется в Perl для остановки выполнения программы в случае ошибки и вывода содержательного сообщения. Вы уже знаете, что простой вызов функции die приводит к появлению сообщения:

Died at ***имя_сценария*** **line** *ххх*

Функции die может передаваться список аргументов, которые будут выводиться вместо стандартного сообщения. Если сообщение не содержит символа перевода строки, то в его конец добавляется текст at ***имя_сценария*** line ***ххх***, например:

```
die "Ошибка при открытии файла";
# Выводится сообщение " Ошибка при открытии файла at имя_сценария   line ххх"
die "Ошибка при открытии файла\n"; # Выводится " Ошибка при открытии файла"
```

В Perl предусмотрена специальная переменная $!, содержащая сообщение об ошибке, возникшей при выполнении последней системной **операции** (например, операции дискового **ввода-вы**вода). В числовом контексте конструкция $! возвращает мало что говорящий номер ошибки. В строковом контексте переменная $! возвращает сообщение операционной системы об ошибке, например:

```
open(MYFILE, "myfile") || die "Ошибка при открытии myfile: $!\n";
```

Если эта функция не сможет открыть файл из-за его отсутствия, будет выведено сообщение Ошибка при открытии myfile: a file or directory in the path does not exist. Как видите, все понятно. Подобные сообщения в сильной степени помогают понять, что произошло, в каком месте программы и в результате выполнения какой операции. Хорошая диагностика неоценима при локализации программных ошибок.



> Не используйте переменную $! для проверки успешности выполнения системной функции. Значение этой переменной определено непосредственно после выполнения системной операции (например, ввода-вывода) и только при неудачном завершении этой операции. Во всех остальных случаях переменная $! может **иметь** совершенно произвольное бессмысленное значение.

Иногда нужно вывести в программе предупредительное сообщение и продолжить ее выполнение. Для генерации предупреждений служит функция warn, аналогичная die, за исключением того, что выполнение программы продолжается:

```
if (! open(MYFILE, "output")) {
    warn "Ошибка при открытии файла output: $!";
} else {
    : # Читаются данные файла...
}
```

Чтение данных из файла

В Perl существует несколько способов чтения файлов, определенных дескрипторами. Самый распространенный заключается в использовании *оператора файлового ввода,* называемого еще *угловым оператором* **(<>)**. Для чтения информации из файла достаточно поместить **его** дескриптор в угловые скобки и присвоить это значение переменной, например:

```
open(MYFILE, "myfile") || die "Ошибка при открытии myfile: $!";
$line=<MYFILE>;          # Чтение файла
```

Угловой оператор в скалярном контексте читает одну строку из файла. Если файл заканчивается, угловой оператор возвращает значение undef.



> *Строкой* в текстовом файле называется последовательность символов, ограниченная специальным признаком конца строки. В UNIX таким признаком является символ перевода строки (ASCII-код 10), в DOS и Windows — последовательность символов возврата каретки и перевода строки (ASCII-коды: 13 и 10). Знамение стандартного признака конца строки может быть изменено в Perl, что позволяет добиться некоторых интересных эффектов. Подробнее об этом речь пойдет на 12-м занятии, "Работа с командной строкой Perl".

Для чтения и вывода содержимого целого файла можно использовать следующий код (в примере предполагается, что MYFILE — открытый дескриптор файла):

```
while (defined($a=<MYFILE>)) {
    print $a;
}
```

Для чтения информации из файла удобно использовать цикл while. Если в цикле while вместо условного выражения используется угловой оператор, Perl автоматически присваивает введенную строку специальной переменной $_ и повторяет цикл, пока файл не закончится:

```
while(<MYFILE>) {
    print $_;
}
```

При этом на оператор while возлагается присваивание введенной строки переменной $_ и проверка признака достижения *конца файла*. Такое интересное поведение случается только в цикле while и лишь тогда, когда условное выражение состоит из углового оператора.



> Не забывайте, что в прочитанных из файла данных, кроме самого текста, содержатся также символы конца строки. Если вам нужен только текст, используйте функцию chomp, позволяющую избавиться от символов конца строки.

В контексте списка угловой оператор читает файл целиком и присваивает его списку. Каждая строка файла присваивается соответствующему элементу списка или массива, как показано ниже:

```
open(MYFILE, "novel.txt") || die "$!";
@contents=<MYFILE>;
close(MYFILE);
```

В этом примере через дескриптор MYFILE считываются все данные из файла novel.txt и присваиваются массиву {(contents. При этом первая строка файла novel.txt присваивается первому элементу массива @contents: $contents [0]. Вторая строка присваивается $contents[l] и т.д.

В большинстве случаев чтение всего файла в массив (если файл не слишком велик) — наиболее простой способ подготовки данных к дальнейшей обработке. При этом можно легко перемещаться по массиву, манипулировать его элементами и пользоваться всем арсеналом средств для работы с массивами. При этом не нужно беспокоиться о том, что содержимое самого файла может быть случайно испорчено, поскольку работа выполняется над копией этого файла. В листинге 5.1 показаны некоторые возможные манипуляции с файлами в памяти.

Листинг 5.1. Инвертирование содержимого файла

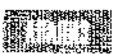
```
1: #!/usr/bin/perl -w
2:
3: open(MYFILE, "testfile") || die "Ошибка открытия файла testfile: $!";
4: @stuff=<MYFILE>;
5: close(MYFILE);
6: # А теперь мы можем выполнить любые манипуляции с файлом.
7: foreach(reverse(@stuff)) {
8:         print scalar(reverse($_));
9: }
```

Поместим в тестовый файл testfile следующий текст:

```
В лесу родилась елочка,
В лесу она росла.
```

Тогда программа из листинга 5.1 выведет:

```
.алсор ано усел В
,акчоле ьсалидор усел В
```



Проведем анализ программы.

- *Строка 1.* В этой строке находится путь к интерпретатору (измените его в соответствии с конфигурацией вашей системы) и ключ -w. Всегда включайте режим вывода предупреждений!
- *Строка 3.* Открывается файл testfile, которому назначается дескриптор MYFILE. Если файл не может быть корректно открыт, выполняется функция die, выводящая сообщение об ошибке.
- *Строка 4.* Все содержимое файла testfile читается в массив @stuff.
- *Строка 5.* Поскольку содержимое файла testfile успешно прочитано, файл можно закрыть и освободить дескриптор MYFILE.
- *Строка 7.* Из массива @stuff создается список с обратным порядком следования элементов — первая строка становится последней и т.д., затем полученный список используется в операторе foreach. Каждая строка этого списка поочередно присваивается переменной $_, после чего выполняется тело цикла foreach.
- *Строка 8.* Изменяется порядок следования символов строки файла (которая находится в переменной $_), после чего выполняется ее печать. Функция scalar в данном случае необходима, так как по умолчанию функция print ожидает получить список, а в контексте списка функция reverse инвертирует порядок следования элементов списка. Поэтому порядок следования символов переменной $_ остался бы неизменным. Функция scalar определяет скалярный контекст для функции reverse, изменяющей порядок следования символов в $_.

Чтение файлов целиком в массив для дальнейшей обработки может быть осуществлено лишь с файлами относительно небольшого размера. Хотя чтение слишком большого файла в память и не запрещено, но может привести к тому, что Perl использует всю доступную память системы.

Если при чтении в массив слишком большого файла или выполнении каких-нибудь других действий доступная для Perl память будет исчерпана, интерпретатор выдаст сообщение об ошибке Out of memory! и прекратит выполнение программы. Если это произойдет при чтении всего файла в массив, вам следует подумать о построчной обработке файла.

Запись в файл

ДЛЯ записи данных в файл сначала нужно открыть сам файл для записи. Синтаксис открытия файла для записи почти такой же, как и для чтения:

```
open(дескриптор, ">путь")
open(дескриптор, ">>путь")
```

Синтаксис первой строки уже знаком нам, за исключением символа > перед путем. Этот символ говорит Perl, что в файл, путь которого указан следом, должны быть записаны новые данные. При этом уже имеющиеся данные в файле стираются и указанный дескриптор открывается для записи. Во втором примере символы » говорят Perl, что файл открывается для записи, но если файл уже существует, новые данные дописываются после имеющихся. Вот примеры:

```
# Новые данные записываются поверх старых, если таковые есть
open(NEWFH, ">output.txt") || die "Ошибка при открытии output.txt: $!";
# Новые данные дописываются к уже существующим.
open(APPFH, ">>logfile.txt") || die " Ошибка при открытии logfile.txt: $!";
```



До сих пор ваши программы на Perl вряд ли могли причинить какой-либо вред. Теперь, когда вы узнали, как записывать данные в файлы, следует очень осторожно использовать эту возможность, чтобы случайно не повредить ценную информацию. В системах с незащищенными системными файлами (Windows 9x и Mac) можно легко повредить операционную систему, случайно уничтожив один из ее файлов. Поэтому тщательно контролируйте, какие файлы вы открываете для записи. Восстановить прежнюю информацию в файле, открытом с помощью префикса >, практически невозможно. Восстановление файлов, случайно открытых с помощью префикса », может оказаться довольно сложным процессом, так что будьте начеку.

После окончания работы с файлом, открытым для записи, вопросом первостепенной важности становится закрытие этого файла и освобождение его дескриптора. Операционная система не переносит данные на диск тут же, как только совершается операция записи, данные вначале записываются в буфер, который периодически сохраняется на диске. Функция close сообщает операционной системе, что запись в файл завершена и данные должны быть помещены в место своего постоянного хранения на диске, например:

```
close(NEWFH);
close(APPFH);
```

После того как файл открыт для записи, поместить в него данные очень просто. Это делается с помощью хорошо вам знакомой функции print. Пока функция print использовалась нами лишь для отображения данных на экране. В принципе эта функция может быть использована для записи данных в любой файл. Синтаксис оператора print, предназначенного для вывода данных в файл, очень прост:

print *дескриптор СПИСОК*

Здесь параметр *дескриптор* — это дескриптор файла, открытого для записи, а *СПИСОК*— список элементов, которые нужно вывести в файл.

Примите во внимание, что синтаксис оператора print не допускает наличия запятой между именем дескриптора и списком. Однако внутри списка запятая используется для разделения элементов списка, как и прежде. Отсутствие запятой между дескриптором и списком говорит Perl, что лексема, следующая за print, — дескриптор файла, а не первый элемент списка. Если вы забудете об этом и поставите запятую, Perl выдаст вам сообщение: No comma allowed after filehandle (если включен режим вывода предупреждений).

Теперь рассмотрим следующий код:

```
open (LOGF, ">>logfile") || die "$!";
if (! Print LOGF "Запись сделана ", scalar(localtime), "\n") {
    warn "Ошибка при записи в файл logfile: $!";
}
close(LOGF);
```

В этом примере файл logfile открывается для добавления информации. Оператор print выводит сообщение в дескриптор LOGF. Значение, возвращаемое функцией print, проверяется, и, если запись не может быть сделана, выводится предупреждение. Затем дескриптор файла закрывается.

Одновременно можно открыть сразу несколько файлов для чтения или записи, как показано в следующем примере:

```
open(SOURCE, "soursefile") || die "$!";
open(DEST, ">destination") || die "$!";
@contents=<SOURCE>;         # "Проглотим" исходный файл
print DEST @contents;       # Запишем его в другой файл
close(DEST);
close(SOURCE);
```

В этом примере выполняется простое копирование файлов. Кстати, можно несколько сократить код, объединив в одном операторе операции чтения и записи:

```
print DEST <SOURCE>;
```

Так как функция print в качестве параметра ожидает передачи списка, оператор <SOURCE> находится в контексте списка. Угловой оператор в контексте списка считывает весь файл, а оператор print выводит его в дескриптор DEST.

Свободные дескрипторы, тестирование файлов и двоичные данные

С точки зрения файловой системы, файлы не обязательно должны быть связаны с данными, хранящимися на диске. Иногда дескриптор может быть связан с объектом операционной системы, например с клавиатурой, сетевым каналом или устройством резервного копирования на магнитную ленту. Кроме того, в файловой системе хранятся так называемые *метаданные,* относящиеся к вашим файлам. Perl может запрашивать файловую систему о размере конкретного файла, времени и исполнителе его последней модификации, а также о том, какого рода данные находятся в файле. В некоторых операционных системах файловые метаданные содержат даже информацию о том, является ли файл текстовым или двоичным.

Свободные дескрипторы

Perl начал свою жизнь в качестве утилиты для UNIX. Поэтому иногда это происхождение проявляется и на других, отличных от UNIX, платформах. Перед запуском программы на Perl для нее автоматически создаются три дескриптора файлов: STDOUT (стандартный выходной поток), STDIN (стандартный входной поток) и STDERR (стандартный поток ошибок). По умолчанию все они связаны с терминалом.

Для считывания введенного вами с клавиатуры текста в Perl используется дескриптор STDIN:

```
$guess=<STDIN>;
```

Для вывода данных на экран терминала предусмотрена функция print, которая по умолчанию использует дескриптор файла STDOUT, например:

```
print "Привет всем!\n";          # то же, что и...
print STDOUT "Привет всем!\n";
```

На 12-м занятии, "Работа с командной строкой Perl", вы узнаете, как изменить имя дескриптора файла, который по умолчанию используется в функции print.

Дескриптор STDERR обычно также связан с терминалом. Он используется для отображения сообщений об ошибках. В UNIX сообщения об ошибках и обычные данные могут быть выведены на различные мониторы. Поэтому традиционный подход заключается в том, чтобы выводить сообщения об ошибках в стандартный поток ошибок STDERR. Например, функции die и warn выводят свои сообщения в STDERR. Если операционная система не поддерживает отдельного потока ошибок, как, например, в случае DOS или Windows, поток STDERR выводится в STDOUT.



Вопрос о перенаправлении вывода и сообщений об ошибках в UNIX выходит за рамки этой книги. В различных оболочках это делается по-своему. Данная тема подробно рассматривается в любой более-менее приличной книге по UNIX.

Работа с бинарными файлами

Некоторые операционные системы, такие как VMS, Atari ST и в особенности Windows и DOS, различают двоичные (бинарные) и текстовые файлы. Это вызывает определенные проблемы, так как Perl не видит между ними отличий. Текстовые файлы состоят из записей, оканчивающихся символами конца строки, называемыми *разделителями записей*. Двоичные файлы — это набор битов, которые должны быть правильно интерпретированы, например изображения, программы и файлы данных.

Когда выполняется запись данных в текстовый файл, Perl рассматривает символ \n как разделитель записей, принятый в данной операционной системе. В UNIX \n преобразуется в ASCII-код 10 (символ LF), в Macintosh и Windows — в ASCII-коды 13 и 10 (CRLF). Это особенность текстовых файлов.

При записи двоичных данных, таких как файлы GIF, EXE, документы MS Word и т.п., преобразование данных не требуется. Поэтому, чтобы ни Perl, ни операционная система не делали подобных преобразований, перед записью двоичных данных в файл необходимо использовать функцию binmode. Она помечает дескриптор файла как двоичный. Функция binmode вызывается после открытия файла, но до того, как будет выполнен ввод или вывод данных. Вот пример:

```
open(FH, "camel.gif") || die "$!";
binmode(FH);         # Дескриптор становится двоичным.
```

```
# Выведем заголовок GIF-файла...
print FH "GIF87a\056\001\045\015\000";
close(FH);
```

После открытия файла функцию **binmode** к его дескриптору можно применить только один раз. При закрытии и повторном открытии двоичного файла придется заново вызвать функцию **binmode**. Использование **binmode** в системах, где не различаются текстовые и двоичные файлы (например, в UNIX), не вызывает никаких действий.

Операторы тестирования файлов

Перед тем как открыть файл, неплохо убедиться, что он действительно существует, проверить, не является ли он каталогом, и не приведет ли это к появлению сообщения об ошибке permission denied. В Perl имеются специальные операторы *тестирования файлов.* Все они имеют похожий синтаксис:

```
-Xдескриптор_файла
-Xпуть
```

Здесь *X* — конкретная операция тестирования, а ***дескриптор_файла*** — тестируемый дескриптор. Файл можно протестировать и без открытия дескриптора. В табл. 5.2 приведены некоторые операторы тестирования.

Таблица 5.2. Часто используемые операторы тестирования файлов

Оператор	Пример	Результат
-r	-r *'файл'*	Истинное значение, если разрешено чтение *'файла'*
-w	-w$a	Истинное значение, если разрешена запись в файл, имя которого содержится в переменной $a
-e	-e *'файл'*	Истинное значение, если *'файл'* существует
-z	-z *'файл'*	Истинное значение, если *'файл'* существует, но он пустой
-s	-s *'файл'*	Возвращает размер *'файла'* в байтах, если тот существует
-f	-i *'файл'*	Истинное значение, если *'файл'* является обычным файлом (не каталогом)
-d	-d *'каталог'*	Истинное значение, если параметр *'каталог'* задает каталог
-T	-T *'файл'*	Истинное значение, если параметр *'файл'* определяет текстовый файл
-B	-B 'файл'	Истинное значение, если параметр *'файл'* определяет двоичный файл
-M	-K *'файл'*	Возвращает количество прошедших дней с момента последней модификации *'файла'*

Полный список операторов тестирования файлов можно найти в документации. Наберите в командной строке perldoc perlfunc и найдите раздел Alphabetical List of Perl Functions.

Операторы тестирования позволяют убедиться в отсутствии файла с данным именем перед созданием нового файла. Их можно использовать для проверки правильности значения, введенного пользователем. С помощью этих операторов можно также проверить наличие некоторого каталога и возможность записи в него. Ниже приведен пример:

```
print "Где будем сохранять данные?";
$filename=<STDIN>;
chomp $filename;
if (-s $filename ) {
     warn "Содержимое файла $file будет потеряно!\п";
     warn "Он был модифицирован ",
          -M $filename, "дней тому назад.\п"
}
```

Резюме

На этом занятии вы изучали открытие и закрытие файлов с помощью функций open и close. Чтение открытого файла выполняется с помощью операторов <> или read, а запись в файл — с использованием оператора print. Еще вы узнали о некоторых особенностях работы операционной системы с файлами и о том, зачем нужна функция binmode.

Кроме того, я надеюсь, что вы не пропустили мимо ушей информацию о безопасном стиле написания программ.

Вопросы и ответы

Мой оператор open не работает по неизвестной причине. Что могло произойти?

Во-первых, проверьте синтаксис выражения с функцией open. Убедитесь, что правильно указано имя файла. Можете даже вывести это имя, перед тем как использовать его в open. Если вы собираетесь использовать файл для записи, не забудьте поставить префикс > перед именем файла, это необходимо. А проверяли ли вы, как выполнилась операция открытия файла с помощью синтаксиса open() j| die "$!";?. Сообщение функции die может сильно облегчить поиск ошибки.

Я выводил данные в файл, но их там не оказалось. Куда делись мои данные?

А вы уверены, что файл открыт правильно? Если вы используете неправильное имя файла, данные могут оказаться не в том файле, в каком вы ожидали. Распространенной ошибкой является использование символов обратной косой черты в пути файла, если путь заключен в двойные кавычки:

```
open(FH, ">c:\temp\notes.txt") || die "$!"; # Ошибка!
```

В этой строке создается файл c:(табулятор)emp(новая строка)otes.txt. Это явно не то, что нужно. Убедитесь также в успешном выполнении функции open. При отключенном режиме выдачи предупреждений Perl "молча" отбрасывает данные, выводимые в файл, который не был успешно открыт.

Я пытался открыть файл с помощью функции open, но получил сообщение permission denied. В чем тут дело?

Perl лишь следует правилам, регламентирующим безопасность в данной операционной системе. Если у вас нет права доступа к соответствующему файлу, каталогу или диску, то Perl ничего не может с этим поделать.

Как организовать посимвольный ввод данных?

Для посимвольного ввода из файла используется функция `getc`. Посимвольный ввод с клавиатуры — гораздо более серьезный вопрос, требующий учета особенностей конкретной операционной системы. После знакомства с модулями Perl на 15-м занятии, "Обработка данных в Perl" и чтения FAQ, речь о котором пойдет в 16-м занятии, "Сообщество Perl", посмотрите пятый раздел FAQ. В нем содержится развернутое объяснение принципов организации посимвольного ввода для различных платформ с многочисленными примерами кода. В этой книге мы не можем их привести.

Как избежать одновременной записи в один и тот же файл со стороны различных программ?

Перед записью нужно выполнить *блокировку* файла. Более подробно эта тема обсуждается на 15-м занятии, "Обработка данных в Perl". Предупреждаем, это довольно сложный вопрос!

Семинар

Контрольные вопросы

1. Чтобы открыть файл data для записи, нужно сделать следующее:

   а) `open(FH, "data", write);`

   б) `open(FH, "data");` а затем просто выполнить печать в `FH`

   в) `open(FH, ">data") || die "Ошибка при открытии data: $!";`

2. Выражение `(-M $file > 1 and -s $file)` истинно, если:

   а) файл `$file` был модифицирован не позднее, чем день назад, и содержит данные;

   б) это выражение не может быть истинным;

   в) запись в `$file` разрешена, и он не содержит данных.

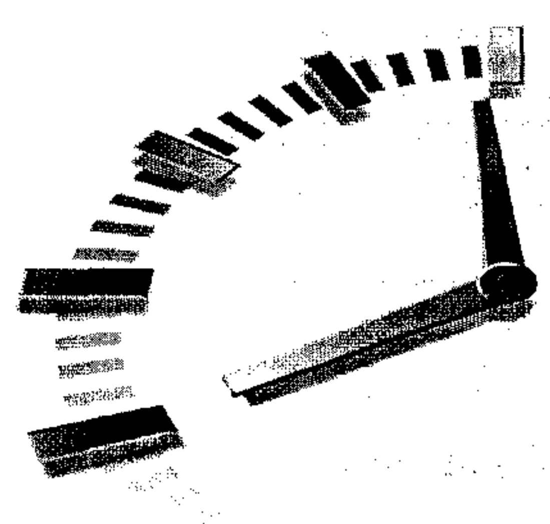
Ответы

1. Правильный ответ — вариант в). У варианта а) неправильный синтаксис, а вариант б) открывает файл для чтения. Вариант в) — то, что нужно, и к тому же хорош для контроля ошибок.

2. Правильным является вариант а). Оператор -M возвращает количество дней, прошедших с момента последней модификации файла (выражение >1 означает, что количество дней должно быть больше одного), а оператор -s возвращает истинное значение лишь для файлов, содержащих данные.

Упражнения

- Модифицируйте программу Hangman, описанную на предыдущем занятии, так, чтобы она читала список возможных слов из файла.



6-й час

Поиск по шаблону

Из предыдущего занятия вы узнали много о чтении данных из файлов. Эта информация и уже известные вам сведения о скалярах, массивах и операторах являются необходимой теоретической базой для создания программ данных. Но, к сожалению, данные в файле не всегда имеют простой формат с пробелами-разделителями, позволяющий воспользоваться простым выражением функции split, а кроме того, в файле могут содержаться строки с ненужными данными, которые должны быть отброшены.

Для этого нужно уметь распознавать в потоке ввода фрагменты текста, соответствующие некоторым шаблонам, извлекать данные с использованием этих шаблонов и, возможно, преобразовать эти данные в более удобную для использования форму. *Регулярные выражения*— одно из средств Perl, предназначенное для выполнения этих задач. Заметим, что в данной книге термины *регулярное выражение* и *шаблон* являются эквивалентными.

Регулярные выражения — это фактически язык в языке. Они предоставляют формальный метод описания шаблонов для поиска. На этом занятии мы рассмотрим только начальные сведения о языке регулярных выражений.



> В документации содержится намного более подробное (и гораздо более трудное для восприятия) описание языка регулярных выражений в Perl {обратитесь к странице perlre). Это настолько сложная тема, что ей посвящены целые книги. Сообщество Perl для дальнейшего изучения регулярных выражений рекомендует книгу Джеффри Фрейдла (Jeffrey E. F. Freidl) *Mastering Regular Expressions* (Sebastopol: O'Reilly, 1997). В ней рассматривается использование регулярных выражений не только в Perl, но именно Perl уделено особое внимание.

Регулярные выражения используются и в других языках программирования, включая TCL, JavaScript, Python и С, а также во многих утилитах операционной системы UNIX. В Perl регулярные выражения представлены достаточно полно, и их знание поможет вам в освоении других языков программирования.

Основные темы этого занятия.

- Как создавать простые регулярные выражения.
- Как использовать регулярные выражения для поиска по шаблону.
- Как редактировать строки при помощи регулярных выражений.

Простые шаблоны

В Perl шаблоны помешаются в оператор поиска по шаблону, который обычно выглядит как т/Л ВОТ пример простого шаблона:

```
ш/Simon/
```

Этот шаблон соответствует последовательности букв **S-i-m-o-n**. Но вот только где он ищет эту последовательность? Ранее вы узнали, что в **Perl** часто используется стандартная переменная $_. Так вот, поиск по шаблону происходит в переменной $_, если не будет указана другая переменная (об этом мы поговорим позже). Итак, предыдущий шаблон ишет последовательность символов **S-i-m-o-n** в скалярной переменной $_.

Если в переменной $_ найдена строка, соответствующая шаблону, оператор m// возвращает истинное значение. Поэтому обычно поиск по шаблону используют в условных выражениях, как показано ниже:

```
if (m/Piglet/) {
        I Здесь строка "Piglet" присутствует в $_
}
```

Внутри шаблона каждый символ соответствует самому себе, если только это не *метасимвол*. Большинство обычных символов (буквы и цифры) соответствуют сами себе. *Метасимволы* — это символы, изменяющие поведение шаблонов. Вот список метасимволов:

```
^ $ ( ) \ | 0 [ { ? . + *
```

Их действие будет объяснено чуть позже. Если же в шаблоне вы хотите использовать метасимвол как литерал, необходимо перед ним поставить обратную косую черту, например:

```
m/Я честно выиграл \$10/; # $ в данном случае "$" не метасимвол, а просто знак
доллара.
```

Оператор поиска по шаблону *обычно* имеет вид m//. Тем не менее в нем можно заменить символы косой черты на любой другой символ, как в следующем примере:

```
if (m,Waldo,) { print "Найдено слово Waldo.\n";}
```

Подобная замена символов часто делается, если в шаблоне есть символы косой черты, что может привести к неправильному определению конца шаблона. Поэтому перед вложенным символом косой черты должен быть символ обратной косой черты, например:

```
if (m/\/usr\/local\/bin\/hangman/) {print "Найдена игра hangman!';}
```

Этот же пример можно переписать в более читабельном виде:

```
if (m:/usr/local/bin/hangman:) {print "Найдена игра hangman!";}
```

В операторе поиска по шаблону можно обойтись и без префикса m, если в качестве *ограничителей* шаблона используются символы косой черты. Так, вместо m/Cheetos/ можно написать /Cheetos/. Обычно, если не нужно заменять символ-ограничитель, так и поступают.

В регулярных выражениях можно использовать переменные. Встретив в регулярном выражении имя переменной, Perl сначала вычисляет значение этой переменной, а затем уже использует получившийся шаблон. Это позволяет создавать регулярные выражения на ходу. В приведенном ниже примере регулярное выражение, используемое в операторе if, считывается с клавиатуры:

```
$pat=<STDIN>; chomp $pat;
$_="Строка, в которой будет выполняться поиск";
if (/$pat/){  f Используется шаблон, введенный пользователем
    print "Строка \"$_\" соответствует шаблону $pat\n";
}
```



Для обозначения регулярных выражений в руководстве и документации часто используются сокращения *RE* или *regexp*. Для ясности мы будем по-прежнему называть их регулярными выражениями.

Правила игры

Прежде чем применять регулярные выражения в Perl, сначала нужно узнать, как они используются. Этих правил немного. Все они возникли не на пустом месте, а имеют определенный смысл. Вот они.

- Обычно поиск по шаблону в строке ведется слева направо.
- Выражение, в котором задан шаблон, возвращает истинное значение (в любом контексте) тогда и только тогда, когда в строке найден текст, полностью соответствующий шаблону.
- Поиск соответствия шаблону выполняется в строке слева направо. Таким образом, первым будет найден текст, расположенный ближе к началу строки. Однако это совсем не означает, что следующие соответствия шаблону найдены не будут. Хотя существуют и исключения...
- Прежде всего программа пытается найти самую длинную строку, соответствующую шаблону. Обнаружив первое совпадение, Perl просматривает всю строку до конца в поиске более длинной фразы, соответствующей заданному шаблону. Для описания этого свойства регулярных выражений придуман специальный термин — *"жадность"* регулярных выражений.

Метасимволы

Во всех примерах этого занятия фрагменты текста, совпавшие с шаблоном, подчеркиваются. Помните, что строка называется соответствующей шаблону, если хотя бы ее часть соответствует этому шаблону. Подчеркивание показывает, какая именно часть совпадает.

Не пропустите следующие разделы и не огорчайтесь, если сразу вы поймете не все, — понимание постепенно придет. Давайте начнем с метасимволов.

Простой метасимвол

Первый метасимвол — это точка (.)• Внутри регулярного выражения точка соответствует любому одиночному символу, кроме символа перевода строки. Например, в шаблоне /p.t/ точка означает любой символ. Этому шаблону соответствуют слова pot, pat, pit, carpet, python и pup_tent. Точка заменяет только один символ. Поэтому слова apt и expect шаблону не подходят, поскольку в первом слове между символами p и t нет никакого символа, а во втором — слишком много символов.

Непечатные символы

Ранее вы узнали, что для включения метасимвола в регулярное выражение В виде литерала перед ним необходимо поставить символ обратной косой черты и тогда метасимвол потеряет свою магическую силу:

```
/\^\$/;   # Знак вставки и знак доллара
```

Если перед обычным символом поставить символ обратной косой черты, он становится метасимволом. Как вы уже знаете из 2-го занятия, "Строительные блоки Perl: числа и строки", некоторые символы в строковых литералах имеют специальное значение, если перед ними стоит символ обратной косой черты. Почти все из них имеют то же значение и в регулярных выражениях, как показано в табл. 6.1.

Таблица 6.1. Специальные символы

Символ	Значение
\n	Перевод строки
\r	Возврат каретки
\t	Табулятор
\f	Прогон страницы

Квантификаторы

Пока мы лишь рассматривали случай, когда символу шаблона соответствует один символ в строке. Например, в шаблоне /Simon/ символ S соответствует *S*, i соответствует *i*, m соответствует *m* и т.д. *Квантификаторы* — это метасимволы, используемые для указания количественных отношений между символами в шаблоне и в искомой строке. Квантификатор может быть поставлен после одиночного символа или после группы символов (о группах мы скоро поговорим).

Простейшим квантификатором является метасимвол +. Он означает, что идущий перед ним символ соответствует нескольким идущим подряд таким символам в строке поиска. Количество символов может быть любым (максимально большим в рамках соответствия шаблону), но должен присутствовать хотя бы один символ. Таким образом, шаблону `/do+g/` будут:

соответствовать	we соответствовать
`Hounddog`	badge (нет буквы *o*)
`hotdog`	doofus (отсутствует буква *g*)
`doogie howser`	Doogie (*D* — это не d)
dooooooooooooogdog	pagoda (буквы d, *o* и *g* идут не по порядку)

Действие метасимвола * схоже с действием +. Метасимвол * указывает, что идущий перед ним символ встречается нуль или более раз. Другими словами, шаблон /t*/ будет искать подряд идущие буквы *t*, но если таких букв вообще нет, поиск все равно будет считаться успешным, т.е. регулярному выражению `/car*t/` будут:

соответствовать	не соответствовать
carted	carrot (лишняя буква о)
cat	carl (буква t в данном шаблоне — обязательная)
carrrt	caart (лишняя *a* не подходит)

Еще более ограниченный диапазон действия у метасимвола ?. Предшествующий ему символ должен встречаться нуль или один раз (но не более того). Так, шаблон /c?ola/ означает, что буква *с* может встретиться один раз или вообще не встретиться. Этот шаблон соответствует любой строке, содержащей символы *ola,* например *cola.*

Различие между метасимволами ? и * состоит в том, что, например, шаблону /c?ola/ соответствуют и *ola,* и *cola,* но не *ccola.* Дополнительная буква с не входит в зону совпадения. Шаблону /c*ola/ будут соответствовать и *cola,* и *ola,* и *ccola,* потому что, в отличие от предыдущего шаблона, для совпадения допускается неограниченное количество подряд идущих букв *с.*

Если возможностей метасимволов +, *, ? вам недостаточно, воспользуйтесь фигурными скобками {} для точного указания количества повторений:

pat{n,m}

Здесь л — минимально допустимое количество повторений, *m* — максимально допустимое количество повторений, а *pat* — символ или группа символов, для которых указывается количество повторений. Один из параметров л или *m* можно опустить, но не оба сразу! Посмотрите на примеры:

```
/x{5,10}/        # x встречается как минимум 5 раз, но не более 10
/x{9,  }/        # x встречается 9 или более раз
/x{0,  4}/       # x встречается не более 4 раз, но может
                 # вообще не встретиться
/x{8}/           # x встречается ровно 8 раз
```

В регулярных выражениях часто используют идиому .*. Ей соответствует все, что угодно, например в шаблоне /first.*last/ — это любые символы, находящиеся между двумя словами. Согласно приведенному шаблону, Perl пытается найти слово *first,* текст за ним и слово *last.* Посмотрите действие шаблона на примере следующих строк:

```
first then last
The good players get picked first, the bad last.
The first shall be last, and the last shall be first.
```

Внимательно посмотрите на третью строку. Совпадение с шаблоном начинается, как и ожидалось, со слова *first*.Далее совпадение включает слово *last* и текст дальше до следующего слова *last.* Здесь метасимвол * следует четвертому правилу, описанному в разделе "Правила игры": находится самая длинная строка, все еще удовлетворяющая шаблону поиска. В случаях, когда требуется отменить действие этого правила, необходимо воспользоваться возможностью *минимального соответствия* в Perl. За подробной информацией по данному вопросу обратитесь к странице руководства perlre.

Классы символов

Другая типичная задача использования регулярных выражений — поиск, при котором подходит любой символ из определенного набора. Для поиска чисел хорошо иметь шаблон, соответствующий любой цифре, для поиска в списке имен типа *Von Beethoven* или *von Beethoven* вам пригодится шаблон "или v, или V".

В регулярных выражениях Perl такая возможность предусмотрена. Речь идет о так называемых *классах символов*. Классы символов заключаются в квадратные скобки [ ]. Во время поиска все символы в классе рассматриваются как один символ. Внутри класса можно задавать диапазон символов (когда такой диапазон имеет смысл), помещая дефис между границами диапазона. В табл. 6.2 приведено несколько примеров.

Таблица 6.2. Примеры использования классов символов

Класс символов	Описание
[abcde]	Любой символ a, *b*, *c*, *d* или e
[a-e]	То же самое, любой символ a, *b*, *c*, *d* или e
[Gg]	Прописная *G* или строчная *д*
[0-9]	Цифра
[0-9]+	Последовательность одной или более цифр
[A-Za-z]{5}	Последовательность из пяти алфавитно-цифровых символов
[*!@#$%&()]	Любой из этих знаков пунктуации

Последний пример наиболее интересен. Из него видно, что внутри символьных классов большинство метасимволов теряют свои значения и становятся обыкновенными символами. Поэтому * — это просто литерал.

Если первым символом класса является знак вставки (^), то значение выражения инвертируется. Другими словами, такому классу соответствует любой символ, *не* входящий в класс, например:

```
/[^A-Z]/;  f Любые символы, кроне символов верхнего
           f регистра
```

Так как в классах символы], ^ и - имеют специальное значение, для их использования в классе существуют определенные правила. Литерал " не должен быть первым символом класса. Перед литералом] должен стоять символ обратной косой черты, например /[aЬc\]]/. Для помещения в класс дефиса (-) достаточно либо поставить его на первую позицию, либо поместить перед ним символ обратной косой черты.

В Perl имеются сокращения для некоторых часто используемых классов. Все эти сокращения состоят из символа обратной косой черты и следующего за ним неметасимвола, как показано в табл. 6.3.

Таблица 6.3. Специальные символьные классы

Шаблон	Описание
	Символ, встречающийся в словах (латинский алфавит), то же, что и [a-zA-Z0-9_]
\W	Символ, не встречающийся в словах, инверсия \w
\d	Цифра, то же, что и [0-9]
\D	Не цифровой символ
\s	Символ пробела, то же, что и [\t\f\r\n]
\S	Символ, не являющийся символом пробела

Ниже приведено несколько примеров:

```
/\d{5}/;       # Пять цифр подряд
/\s\w+\s/;     \ Группа символов слов, окруженных
               # символами пробела
```

Но будьте внимательны! Последний шаблон не всегда соответствует слову, например символ подчеркивания, окруженный пробелами, также будет считаться словом. Кроме того, с помощью последнего шаблона будут найдены не все слова, а только те, которые окружены пробелами. При этом слова типа don't не будут найдены из-за апострофа. Ниже вам будет предложен гораздо лучший шаблон для поиска слов.

Группировка и альтернация

В регулярных выражениях можно объединять несколько шаблонов, так чтобы найденная строка соответствовала хотя бы одному из них. Это полезно, если, к примеру, необходимо проверить строку на наличие в ней слов *dogs* или *cats*. Для решения подобной проблемы служит операция *альтернации,* которая в регулярных выражениях задается символом |, например:

```
if(/dogs|cats/) {
   print "В строке \$_ говорится о домашних животных\n";
}
```

Альтернация — полезная вещь, но не всегда удобная, если нужно найти большое количество похожих слов. Пусть вам нужно найти одно из слов *frog*, *bog*, *log*, *flog* или *clog,* а выражение /frog|bog|log|flog|clog/ кажется вам слишком громоздким. Тогда нужно воспользоваться альтернацией только для начала строки. Вы можете попытаться использовать такой шаблон:

```
/fr|b|l|fl|clog/;     # Ничего не выйдет!
```

С помощью данного шаблона не удастся добиться требуемого эффекта, потому что Perl не знает, что альтернация касается начальной части строки. Для решения этой проблемы необходимо воспользоваться группировкой регулярных выражений Perl с помощью символов скобок {).

```
/(fr|b|l|fl|cl)og/;
```

Допускается вложение скобок, а следовательно, и вложение групп. Предыдущий пример можно переписать и так: /(fr|b|l|(f|c)l)og/.

В контексте списка оператор поиска по шаблону возвращает список частей найденного выражения, соответствующих группам шаблона. Каждая группа возвращает соответствующее ей значение в список, а если групп в шаблоне нет — возвращается просто 1. Вот пример:

```
$_="apple is red";
($fruit, $color)=/(.*)\sis\s(.*)/;
```

В данном случае шаблон соответствует произвольной последовательности символов (она задана в виде первой группы), за которой расположен пробел, слово *is,* еще один пробел, а затем другая произвольная последовательность символов (она задана в виде второй группы). Значения частей строки поиска, соответствующие группам регулярного выражения, присваиваются элементам списка $fruit и $color, расположенного в левой части оператора присваивания.

Анкеры

Последние два метасимвола (могу поспорить, вы уже думаете: "Когда же они наконец закончатся?!") — это анкеры. С их помощью можно указать, в каком месте строки (в начале или в конце) должно быть найдено соответствие с шаблоном.

' Первый из этих анкеров — символ вставки (^). Этот символ, помешенный в начале регулярного выражения, говорит о том, что соответствие шаблону должно быть найдено в начале строки. Например, /"video/ соответствует слову *video,* но только если оно находится в начале строки.

Его двойник — символ доллара ($). Этот символ, помещенный в конец регулярного выражения, говорит о том, что соответствие шаблону должно быть найдено в конце строки. Например, /earth$/ соответствует слову *earth,* но только если оно находится в конце строки. Ниже приведено несколько примеров (табл. 6.4).

Таблица 6.4. Пример использования анкеров

Шаблон	Описание
/"Help/	Находит строки, начинающиеся с *Help*
/"Frankly. *darn$/	Находит строки, начинающиеся со слова *Frankly* и заканчивающиеся словом *darn.* Между этими словами может быть все, что угодно
/^hysteria$/	Находит строки, состоящие только из слова *hysteria*
/^$/	Находит пустые строки
/^/	Находит начало любой строки. Шаблон /$/ имеет похожее действие

Подстановка

Поиск по определенному шаблону во введенных строках — это только полдела. Часто требуется модифицировать найденные данные. Для этого можно воспользоваться оператором подстановки s///,хотя это далеко не единственный способ. Синтаксис этого оператора выглядит так:

```
s/шаблон_поиска/строка_замены/;
```

По умолчанию оператор подстановки ищет строку, соответствующую шаблону в переменной $_, и заменяет ее на *строку_замены.* Оператор подстановки возвращает количество выполненных замен или 0, если ни одной замены не было сделано. Вот пример:

```
$_="Our house is in the middle of our street";
s/middle/end/;     # Сейчас: Our house is in the end of our street
s/in/at/;          # Сейчас: Our house is at the end of our street
if (s/apartment/condo/) {
                   # Этот код не будет выполняться, см. примечание.
}
```

В этом примере подстановки произошли, как вы и ожидали. Слово middle было изменено на *end,* а *in* — на at. Блок кода оператора if не выполняется, так как в переменной $_ отсутствует слово apartment, поэтому подстановка невозможна.

В операторе подстановки кроме обратной косой черты можно использовать и другие **символы-ограничители**. Для этого просто поместите требуемый символ-ограничитель сразу же после префикса s, как показано ниже:

```
s#street#avenue#;
```

Упражнение: очистка входных данных

Подстановка данных вслепую, без проверки правильности выполнения команды (как в примере из предыдущего раздела), — обычная практика при обработке массивов данных. При этом программа считывает исходные данные с клавиатуры или файла и форматирует их в надлежащем виде для последующей обработки. В листинге 6.1 приведен простой пример программы демонстрации обработки данных, которая вычисляет массу тела на Луне по его Земной массе.

Наберите в текстовом редакторе программу из листинга 6.1 и сохраните ее под именем Moon. Сделайте файл программы выполняемым с помощью инструкций, приведенных на 1-м занятии, "Начало работы с Perl".

Пример выходных данных этой программы приведен в листинге 6.2.

Листинг 6.1. Программа вычисления массы тела на Луне

```
1: #!/usr/bin/perl -w
2:
3: print "Введите массу тепа:";
4: $_=<STDIN>; chomp;
5: s/^\s+//;   # Удаляет пробелы в начале строка
6:
7: if (m/(lbs?|кг?|килограмм?|фунт?)/i) {
8:         if (s/\s*(lbs?|фунт?).*//i) {
9:                 $_*=0.4536;
10:        } else {
11:                s/\s*(кг?|килограмм?).*//;
12:        }
13:}
14:print "Масса тепа на Луне: ", $_*.16667, " кг\n";
```

Листинг 6.2. Пример вывода программы Moon

```
1: $ perl Moon
2: Введите массу тела:4 кг
3: Масса тела на Пуне: 0.66668 кг
4: $ perl Moon
5: Введите массу тела:6 фунт
6: Масса тепа на Пуне: 0.453609072 кг
```



Проведем анализ программы.

- *Строка 1.* В этой строке находится путь к интерпретатору (измените его в соответствии с конфигурацией вашей системы) и ключ -w. Всегда включайте режим выдачи предупреждений!

- Строки *3—4.* Здесь пользователь вводит свой вес, и функция chomp удаляет символ новой строки из переменной `$_`. Напоминаем, что если не указана другая переменная, то по умолчанию функция chomp использует $_.
- *Строка 5.* Шаблон `/^\s+/` находит символы пробелов в начале введенной строки. Поскольку строка замены отсутствует, то найденные пробелы просто удаляются.
- *Строка* 7. Проверяется наличие во введенной строке допустимой единицы измерения.
- *Строки 8—9.* Шаблон s/\s*(lbs?|фунт?) .*//i находит во входной строке слова *lbs* или *фунт* с возможными символами пробела перед ними (при этом учитывается падеж или множественное число). Например, слова *фунт* (как с пробелом впереди, так и без) или lbs будут удалены из входной строки. При этом значение переменной $_ умножается на `0.4536` для перевода фунтов в килограммы.
- *Строка 11.* В противном случае из переменной $_ удаляются слова *кг* или *килограмм,* а также окружающие их пробелы.
- *Строка 14.* Уже переведенная в килограммы масса тела, находящаяся в переменной $_, умножается на 1/6 и выводится на печать.

Дополнительная информация о регулярных выражениях

Выше мы рассмотрели основные способы обработки текста, находящегося в переменной $_, с помощью оператора подстановки. Теперь вас ждет новая порция информации по функциональным возможностям регулярных выражений. Эффективное использование регулярных выражений предполагает возможность работы не только с $_, но и с другими переменными, использование сложных подстановок, а также применение функций Perl, имеющих отношение к регулярным выражениям.

Работа с другими переменными

В предыдущей программе введенная пользователем масса хранилась в переменной $_, поэтому все поиски по шаблону и подстановки выполнялись с этой переменной. Переменная $_ — не самое удобное место для хранения массы тела. Во-первых, это не очень понятно для начинающих программистов, а во-вторых значение переменной $_ может быть неожиданно изменено в другом месте программы.



Вообще длительное хранение чего либо в $_— это игра с огнем, поскольку многие из операторов **Perl** по умолчанию используют эту переменную и модифицируют ее. В **Perl** $_ считается переменной общего пользования и длительное хранение в ней каких-либо данных, как правило, приводит к ошибкам. Изучив материал 8-го занятия, **"Функции"**, вы сами в этом убедитесь.

В программе, приведенной в листинге 6.1, лучше ввести новую переменную $weight. Для выполнения с ней операций поиска и замены необходимо "привязать" эту переменную к соответствующим операторам. Для этого используется оператор привязки =~:

```
$veight="185 lbs";
$weight=~/ lbs//; # Подстановка выполняется в переменной $weight
```

Оператор =~ не производит никакого присваивания, он просто определяет, что оператор справа действует на переменную слева. Значение всего выражения такое же, как и при использовании переменной $_, как можно видеть из примера:

```
$poem="One fish, two fish, red fish";
$n=$poem=~m/fish/; # $n имеет истинное значевве,
                   # если в $poem есть слово fish
```

Модификаторы и многократный поиск

До сих пор наши регулярные выражения были *чувствительны к регистру.* Это значит, что символы верхнего и нижнего регистров воспринимались ими как разные. Для поиска слов без учета регистра букв можно использовать шаблоны, подобные этим:

```
/[Mm][Aa][Cc][Bb][Ee][Tt][Hh]/;
```

Как видите, это довольно громоздкий способ, часто приводящий к ошибкам из-за неправильного набора пар букв. Существует гораздо лучший способ. В операторах подстановки (s///) или поиска по шаблону (ra//) предусмотрен режим, в котором не учитывается регистр букв в искомых словах. Для этого после шаблона нужно поместить суффикс i, например:

```
/macbeth/i;
```

Этот оператор найдет слово *Macbeth* в прописном, строчном или каком-нибудь иномсмешанномварианте(*MaCbEtH*).

Кроме суффикса i, в операторах поиска и подстановки можно использовать модификатор д, который задает режим глобального поиска. Поиск (или подстановка) при этом будут выполняться многократно по всей строке: сначала находится первое слева совпадение и выполняется первая подстановка, а затем поиск и замена продолжаются до конца строки.

В контексте списка оператор поиска с модификатором g возвращает список, элементы которого соответствуют группам регулярного выражения, например:

```
$_="One fish, two frog, red fred, blue foul";
@F=m/\W(f\w\w\w)/g;
```

Шаблон состоит из символа-разделителя (любого не текстового символа), буквы *f* и трех любых текстовых символов. Буква *f* и три символа объединены в группу с помощью скобок. После вычисления выражения в массиве @F будет содержаться четыре элемента: *fish, frog,* fred и *foul.*

В скалярном контексте оператор поиска с модификатором *д* проходит всю строку и при каждом совпадении возвращает истинное значение. Если больше совпадений не найдено, то возвращается ложное значение. Вот пример:

```
$letters=0;
$phrase="What's my line?";
vhile{$phrase=~/\w/g>  {
    $letters++;
}
```

Здесь оператор поиска (//) с модификатором g находится в скалярном контексте, поскольку он используется в условном выражении while. Шаблон предназначен для поиска текстового символа. Цикл while продолжается (и переменная $letter увеличивается) до тех пор, пока оператор поиска не возвращает ложное значение. После окончания цикла переменная $letters будет иметь значение 11.

> Значительно более эффективный метод подсчета символов будет рассмотрен на 9-м занятии, "Дополнительные функции и операторы".

Обратные ссылки

ЕСЛИ В регулярных выражениях Perl используются скобки, части искомой строки, соответствующие фрагментам в скобках, запоминаются в специальных переменных $1 (первый фрагмент в скобках), $2 (второй), $3, $4 и т.д. Вот пример:

```
/(\d{3})-(\d{3})-(\d{4})/
 └──┬──┘ └──┬──┘ └──┬──┘
    $1      $2      $3
```

Этот шаблон служит для поиска правильно отформатированных телефонных номеров, принятых в США и Канаде, например таких как 800-555-1212. Каждая порция номера запоминается в специальных переменных $1, $2 и $3. Эти переменные могут быть использованы в программе, например:

```
if (/(\d{3>)-(\d{3>)-(\d{4>)/)  {
    print "Код региона — $1";
}
```

Кроме программы, специальные переменные $1, $2 и т.д. могут использоваться в строке замены оператора подстановки, например:

```
s/(\d{3})-(\d{3})-(\d{4})/Код региона $1, телефон $2-$3/;
```

Но будьте внимательны, эти переменные модифицируются при каждом успешном поиске, независимо от использования в регулярном выражении скобок. Кроме того, значения этих переменных *устанавливаются тогда и только тогда, когда строка полностью соответствует шаблону.* Приняв это во внимание, посмотрите на пример:

```
m/(\d{3})-(\d{3})-(\d{4})/;
print "Код региона — $1";
# Плохой стиль, поскольку не принимается во внимание
$ вероятность неудачного поиска
```

В этом примере переменная $1 используется без проверки успешности поиска по шаблону. В случае неудачного поиска это может привести к ошибке в программе.

Новая функция: grep

Поиск в массиве по шаблону — одна из наиболее распространенных операций Perl. Например, вы считываете весь файл в массив и хотите знать, в каких строках файла встречается определенное слово. Как раз для подобных ситуаций в Perl имеется функция grep. Вот ее синтаксис:

```
grep выражение, список
grep блок список
```

Функция grep проходит каждый элемент *списка* и выполняет для него указанное *выражение* или **блок**. Внутри *выражения* или *блока* в качестве очередного элемента списка выступает переменная $_. Если выражение истинно, данный элемент возвращается функцией grep. Ниже приведен пример:

```
@dogs=qw(greyhound bloodhound terrier mutt chihuahua);
@hounds=grep /hound/, @ dogs;
```

Здесь каждый элемент массива @dogs поочередно присваивается переменной $_, затем в этой переменной производится поиск по шаблону /hound/. Каждый из элементов, для которого это выражение истинно, возвращается функцией grep и помещается в массив @hounds.

Существует два важных момента. Первый заключается в том, что переменная $_ ссылается на реальный элемент списка и модификация $_ изменяет значение этого элемента, например:

```
@hounds=grep s/hound/hounds/, @dogs;
```

После выполнения этого кода массив @hounds будет содержать элементы *greyhounds* и *bloodhounds* (обратите внимание на *s* в конце этих слов). Исходный массив @dogs также изменяется из-за изменения значения переменной $_. Теперь в нем будут содержаться элементы *greyhounds*, *bloodhounds, terrier, mutt* и *chihuahua*.

Другой важный момент, о котором программисты часто забывают, состоит в том, что в функции grep необязательно должен использоваться оператор поиска или подстановки. Вместо него может быть использован любой оператор. В следующем примере выбираются породы собак, в названии которых содержится более восьми букв:

```
@longdogs=grep length($_)>8, @dogs;
```



Функция grep получила свое имя от команды UNIX, используемой для поиска по шаблону в файлах. Это настолько полезная команда, что ее название стало именем нарицательным в мире UNIX. Для представителей этого мира выражение "grep-нуть книгу" — означает пролистать книгу в поисках нужного материала.

Родственная grep функция map имеет аналогичный синтаксис. Отличие между ними состоит в том, что map возвращает значение блока или выражения, а не значение переменной $_. С помощью функции map из одного массива можно легко образовать другой, например:

```
@words=map {split ' ', $_} @input;
```

В этом примере каждый элемент массива @input, передаваемый в блок как $_, разбивается по словам функцией split. В качестве символа-разделителя используется пробел. Это означает, что каждый элемент массива @input преобразуется в список слов, которые помещаются в массив Swords.

Резюме

На этом занятии вы узнали, что такое регулярные выражения, как их составлять и для чего они используются в Perl. Регулярные выражения состоят из обычных символов и метасимволов. Обычные символы, как правило, соответствуют самим себе, а метасимволы изменяют значения обычных символов (или друг друга). Регулярные выражения используются для поиска строк по шаблону и для подстановок.

Вопросы и ответы

Почему с помощью шаблона /\W(\w)+\W/ можно найти не все слова в строке, а только те, которые находятся в середине строки?

Вы ищете слово, окруженное символами-разделителями. Первое слово строки, если оно расположено в ее начале, не имеет перед собой никакого символа-разделителя.

Какая разница между m// и //? Я не нахожу никакой.

Действительно, между ними нет почти никакой разницы. Буква m нужна лишь при использовании символов-разделителей шаблона, отличных от /, как, например, в m!*шаблон*!.

Я проверяю, правильно ли пользователь вводит число, а шаблон /\d*/ не работает. У него всегда истинное значение!

Поиск по данному шаблону с квантификатором * всегда успешен, неважно, найдены 2 цифры, 100, 1000 или вообще ни одной. Чтобы гарантировать наличие хотя бы одной цифры, нужно использовать шаблон /\d+/.

Семинар

Если вы хоть как-то разобрались в этих регулярных выражениях и шаблонах, проверьте свои знания при помощи простых тестов.

Контрольные вопросы

1. ЕСЛИ строки имеют формат x=y, какое выражение поменяет местами левые и правые части равенств?

   а) s/{.+)=(.+)/$2=$1/;

   б) s/(*)=(*)/$2=$1/;

   в) s/(.*)=(.*)/$2$1/;

2. Какое значение примет переменная $2 после выполнения этого кода?

```
$foo="Star Wars: The Phantom Menace";
$foo=~/star\s((Wars): The Phantom Menace)/;
```

   а) $2 не установлена, так как поиск завершился неудачей;

   б) *Wars;*

   в) *Wars: The Phantom Menace.*

3. Чему соответствует шаблон m/^[-+]?[0-9]+(\.[0-9]*)?$/?"

   а) датам, представленным в формате мм-дд-гггг;

   б) числам, таким как 45, 15.3 или -0.61;

   в) суммам: 4+12 или 89+2.

Ответы

1. Правильный ответ — вариант а). Вариант в) не содержит символа равенства в строке замены. Вариант б) совершенно неправильный — перед * должен быть какой-то символ. Вариант а) полностью подходит.

2. Правильным ответом является вариант а). Поиск не удался, потому что слово star написано строчными буквами, а оператор поиска не содержит модификатора i. Именно поэтому нужно всегда проверять, удался ли поиск, перед использованием переменных $1, $2 и остальных, подобных им. Если

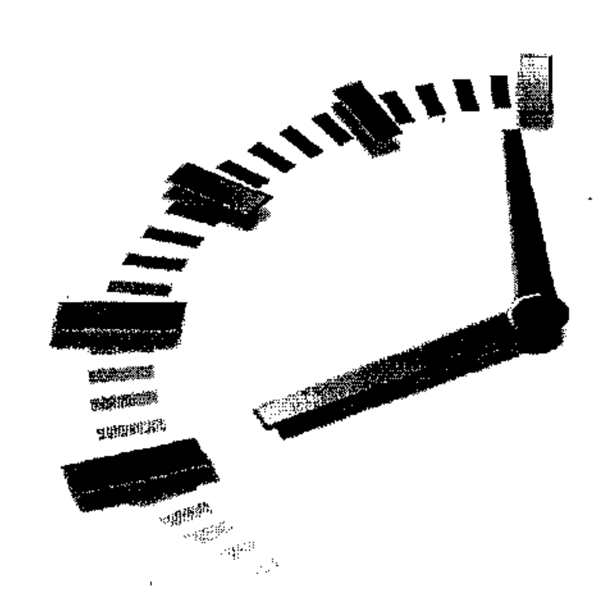
бы в операторе присутствовал модификатор i или слово star было написано с прописной буквы, правильным был бы ответ б).

3. Правильный ответ — вариант б). Шаблон ищет в начале строки необязательные символы + или - затем одну или несколько цифр, необязательную десятичную точку, еще цифры и следуемый за ними конец строки. Это соответствует правильно отформатированным числам.

Упражнения

- Постарайтесь составить шаблон, соответствующий стандартному формату времени. Под этот шаблон должны подходить, например, такие строки: 12:00am, 5:00pm, 8:30AM. А эти строки не должны восприниматься шаблоном — 3:00, 2:60am, 99:00am, 3:0pm.
- Напишите короткую программу, которая делает следующее.

1. Открывает файл.
2. Считывает все его строки в массив.
3. Извлекает все слова из каждой строки.
4. Находит все слова с четырьмя и большим количеством смежных согласных букв, например слова thoughts или yardstick.



7-й час

Хэши

Хэши — третий основной тип данных в Perl. Первый — это скаляры, предназначенные для хранения одиночных значений произвольного типа, и размер. Второй — массивы, представляющие собой коллекции скаляров. Массив может содержать любое количество элементов, но поиск в массиве определенного элемента зачастую связан с последовательным перебором всех элементов массива.

Хэши — еше один вид коллективных данных. Как и массивы, хэши представляют собой коллекцию скаляров. Разница между ними состоит в том, что доступ к элементам хэша осуществляется по имени, а не с помощью числовых индексов, как в массивах. Элементы хэша состоят из двух частей — *ключа* и *значения*. Ключ идентифицирует элемент хэша, а значение содержит данные, связанные с этим ключом. Такая взаимосвязь описывается термином *пара ключ-значение*.

Подобные структуры данных как нельзя лучше подходят для многих приложений. Например, для хранения информации о всех водительских лицензиях в некотором регионе удобно в качестве ключа использовать их номера, поскольку номер лицензии конкретного водителя уникален. С этими ключами можно связать данные о водителе: тип лицензии, адрес, возраст и т.д. Каждая водительская лицензия будет представлена отдельным элементом хэша с номером и необходимой информацией, заданной в виде пары ключ-значение. Вот еще примеры других хэш-подобных структур данных: инвентарные книги, медицинские карточки, телефонные счета, дисковая файловая система, музыкальные коллекции на компакт-дисках, библиотечные каталоги и многое, многое другое.

Хэши в Perl могут содержать произвольное количество элементов, во всяком случае, насколько это позволяют ресурсы системной памяти. Хэши изменяют свои размеры по мере того, как к ним добавляются новые элементы или удаляются старые. Доступ к индивидуальному элементу хэша происходит очень быстро и не сильно замедляется с ростом хэша. Это означает, что Perl комфортно чувствует себя и с хэшем из 10, и из 100 000 элементов. Ключи хэша могут быть произвольной длины, т.е. они являются обыкновенными скалярами. Данные также могут быть произвольной длины.

Раньше хэши в Perl и в других языках программирования назывались *ассоциативными массивами*. Этот термин означал, что ключи элементов связаны с их значениями. Но поскольку программисты на Perl — люди немногословные, они придумали более короткий термин для обозначения ассоциативного массива — хэш.

Переменные-хэши обозначаются в Perl символом процента (%). Их имена не пересекаются с именами массивов и скаляров. Например, в программе может существовать хэш %a, массив @a и скаляр $a. Каждое из этих имен относится к самостоятельной переменной.

Основные темы этого занятия.

- Создание хэша.
- Вставка и удаление элементов хэша.
- Использование хэша для обработки массивов.

Наполнение хэша

Индивидуальные элементы хэша создаются путем присвоения им значений, как и в случае массивов. Вот типичный пример:

```
$Authors{'Dune'}='Frank Herbert';
```

В этом примере присваивается значение элементу хэша %Authors. Ключ этого элемента — слово Dune, а данные — имя Frank Herbert. Этот оператор присваивания создает связь в хэше между словами Dune и Frank Herbert. С переменной $Authors{'Dune'} можно обращаться так же, как и с любым скаляром: ее значение можно передавать в функции, модифицировать с помощью операторов, распечатывать и переопределять. Не забывайте, что при изменении элемента хэша модифицируется значение этого элемента, а не сам хэш.

Почему в нашем примере используется синтаксис $Authors{}, а не %Authors{}? Дело в том, что специальный символ перед именами хэшей, как и в случае массивов, появляется только тогда, когда они рассматриваются как единое целое. Для доступа к индивидуальному элементу хэша используется символ доллара ($), свидетельствующий о том, что это скалярное значение, а фигурные скобки обозначают принадлежность к хэшу. В Perl конструкция $Authors{'Dune'} соответствует одному скалярному значению, в данном случае Frank Herbert.

Хэш, содержащий только один ключ, не представляет собой никакой ценности. Для помещения в хэш нескольких значений необходимо выполнить серию операторов присваивания, например:

```
$food{'apple'}='fruit';
$food{'pear'}='fruit';
$food{'carrot'}='vegetable';
```

Для того чтобы сократить эту операцию, для инициализации хэша можно использовать список. Список должен содержать пары ключей и значений, как показано ниже:

```
%food=('apple', 'fruit', 'pear', 'fruit', 'carrot', 'vegetable');
```

Этот пример напоминает инициализацию массивов, речь о которой шла на 4-м занятии, "Укладка строительных блоков: списки и массивы". Ниже вы узнаете, что во многих контекстах хэши ведут себя как массивы.

При инициализации хэша большим списком бывает трудно определить, где ключ, а где значение. В Perl имеется специальный оператор запятая-стрелка =>. Учитывая тот факт, что символы пробела игнорируются в программах на Perl, инициализацию хэша можно переписать следующим образом:

```
%food=( 'apple' => 'fruit',
        'pear' => 'fruit',
       'carrot' => 'vegetable',
      );
```

Программисты на Perl, хорошо известные своей любовью ко всевозможным упрощениям, придумали еще два дополнительных способа инициализации хэша. Предполагается, что левая часть оператора => — простая строка, которую не нужно заключать в кавычки. К тому же ключ хэша, состоящий из одного слова в фигурных скобках, также можно не заключать в кавычки. Поэтому предыдущие инициализации можно упростить:

```
$Books{Dune}='Frank Herbert';
%food=( apple => 'fruit', pear => 'fruit', carrot => 'vegetable' );
```



Оператор запятая-стрелка называется так потому, что действует как запятая (разделяет элементы списка) и похож на стрелку.

Получение данных из хэша

ДЛЯ извлечения индивидуального элемента из хэша нужно набрать символ $, имя хэша и ключ элемента, например:

```
%Movies= ( 'The Shining' => 'Kubrick',
      'Ten Commandments' => 'DeMille',
               Goonies => 'Spielberg');
print $Hovies{'The Shining'};
```

Этот фрагмент кода распечатывает элемент The Shining хэша %Movies. В результате будет выведено слово Kubrick.

Иногда требуется выполнить проверку всех элементов хэша. Если все ключи хэша известны, можно получить доступ к любому элементу хэша. Но обычно неудобно выполнять общие операции с элементами хэша, явно указывая имена ключей, поскольку все они могут быть неизвестны или их может быть слишком много для перечисления.

Для получения списка ключей хэша используется функция keys. Ключи хэша не упорядочены, поэтому функция keys также возвращает список ключей в неупорядоченном виде. Для вывода всех фильмов из нашего хэша используется следующий код:

```
foreach $film (keys %Movies)  {
    print "$film\n";
}
```

В этом примере переменной $film поочередно присваиваются значения всех элементов списка, возвращаемого выражением keys %Movies. Если кроме названий фильмов нужно распечатать имена режиссеров, используйте следующий код:

```
foreach $film (keys %Movies) {
    print "Фильм $film был снят режнссером $Movies{$film}.\n";
}
```

После выполнения этого кода будут выведены следующие строки:

```
Фильм Ten Commandments был снят режиссером DeMille.
Фильм The Shining был снят режнссером Kubrick.
Фильм Goonies был снят режиссером Spielberg.
```

Так как переменная $film содержит значение ключа хэша, конструкция $Movies{$film} возвращает значение элемента хэша с этим ключом. Для демонстрации взаимосвязи ключа и значения элемента хэша полезно их распечатать вместе. Однако следует иметь в виду,

что ваш результат может отличаться от приведенного в книге порядком следования элементов, поскольку функция keys возвращает ключи в неупорядоченном виде.

Кроме keys, в Perl есть также функция values, предназначенная для доступа к значениям всех элементов хэша. Извлечение значений отдельно от ключей не позволяет узнать, с каким ключом связано каждое конкретное значение. Функция values возвращает значения хэша в том же порядке, что и keys возвращает ключи. Вот пример:

```
@Directors=values %Movies;
@Films=keys %Movies;
```

Здесь элементы `@Directors` и `@Films` с одинаковым индексом образуют пару ключ-значение хэша fcMovies. Имя режиссера в `$Directors[0]` соответствует названию фильма `$Films[0]` и т.д.

Часто нужно извлечь индивидуальный элемент хэша не по ключу, а по его значению. Для этого существует специальный метод доступа, называемый *инверсией хэша*. Он заключается в том, что ключи и значения меняются местами, т.е. все ключи становятся значениями, а все значения становятся ключами:

```
%Movies= ( 'The Shining' => 'Kubrick',
      'Ten Commandments' => 'DeMille',
                 Goonies => 'Spielberg');
%ByDirector=reverse %Movies;
```

Как этот код работает? Функция reverse рассматривает хэш как обычный список:

```
( 'The Shining', 'Kubrick',
  'Ten Commandments', 'DeMille',
  'Goonies', 'Spielberg' )
```

Затем порядок следования элементов этого списка изменяется на обратный, в результате чего получается следующий список:

```
( 'Spielberg', 'Goonies',
   'DeMille' , 'Ten Commandments',
    'Kubrick', 'The Shining' )
```

Обратите внимание, что внутри пар ключ-значение произошли перестановки — значения теперь находятся впереди. После присвоения получившегося списка хэшу `%ByDirector` мы получаем хэш, практически идентичный исходному, за исключением того, что значения исходного хэша стали ключами нового, а ключи исходного — значениями нового. Будьте осторожны с подобной операцией, если некоторые элементы исходного хэша имеют одинаковые значения. Если значения, которые становятся ключами, не уникальны, в новом хэше будет меньшее количество элементов, чем в исходном. Как только в новом хэше будет встречаться уже имеющееся значение, старый ключ будет заменяться новым.

Списки и хэши

При обсуждении вопроса инициализации хэша отмечалась близкая связь хэшей и массивов. Когда хэш используется в контексте списка, Perl воспринимает его в виде обычного списка ключей и значений. Этот список может быть присвоен массиву, как и любой другой список:

```
%Movies= ( 'The Shining' => 'Kubrick',
      'Ten Commandments' => 'DeMille',
                 Goonies => 'Spielberg');
@Data=%Movies;
```

После выполнения этого кода массив @Data будет содержать шесть элементов. Причем элементы с четными индексами (включая нуль) соответствуют именам режиссеров, а элементы с нечетными индексами — названию фильмов. С этим массивом можно выполнить любую операцию, возможную с массивами, и затем присвоить получившийся массив хэшу %Movies:

```
%Movies=@Data;
```



Perl хранит элементы хэша в удобном для обработки порядке, который с точки зрения пользователя является случайным. Последовательность ввода ключей хэша не запоминается и выводятся они в произвольном порядке. Для упорядоченного вывода элементов хэша нужно или отсортировать их (см. ниже раздел "Практическое применение хэшей"), или запомнить порядок, в котором вводились ключи (см. раздел "Вопросы и ответы" в конце этого занятия).

Массивы и хэши имеют сходства и в другом. Для копирования хэша достаточно присвоить один хэш другому:

```
%New_Hash=%Old_Hash;
```

Когда вы помещаете %Old_Hash в правую часть оператора присваивания, там, где обычно Perl ожидает встретить список или массив, Perl преобразует хэш в список, используемый в дальнейшем для инициализации %New_Hash. Подобным образом вы можете комбинировать хэши и манипулировать ими, как списками:

```
%Both=(%First, %Second);
%Additional=(%Both, key1 => 'value1', key2 => 'value2');
```

Первый список объединяет два хэша, %First и %Second в третий хэш %Both. Если некоторые ключи %First присутствуют в %Second, пара ключ-значение второго хэша заменяет пару ключ-значение первого. Во второй строке %Both представлен списком пар ключ-значение. Кроме него в скобках есть еще две пары ключ-значение. Полученный полный список используется для инициализации хэша %Additional.

Дополнительная информация о хэшах

ЕСЛИ ВЫ НОВИЧОК В Perl, некоторые операции с хэшами будут для вас далеко не очевидны. Из-за специфической природы хэшей для некоторых часто встречающихся операций понадобятся новые функции, в которых не было необходимости при работе со скалярами и массивами.

Проверка ключей хэша

Чтобы проверить, существует ли некоторый ключ в хэше, казалось бы, можно использовать следующий синтаксис:

```
if ( $Hash{keyval} )  { t Этот код неправильный
:
}
```

Этот пример не будет работать, поскольку в нем нет проверки, действительно ли keyval является ключом хэша, вместо этого проверяется значение, соответствующее ключу keyval. А если написать так:

```
if { defined $Hash{keyval} )  { i Опять неправильный код
:
}
```

И это не совсем то, что нужно. В нем проверяется существование данных, связанных с ключом keyval, а не самого ключа. А ведь вполне допускается связывать с ключом хэша значение undef:

```
$Hash{keyval}=undef;
```

Проверка данного элемента выдаст ложное значение, потому что проверяется не наличие ключа, а связанное с ним значение. Итак, как же сделать проверку ключей корректно? Для этого в Perl есть специальная функция exists. Функция exists проверяет наличие указанного ключа в хэше и возврашает либо истинное значение (если ключ существует), либо ложное (в противном случае):

```
if { exists $Hash{keyval} )    {  # Теперь правильно!
:
}
```

Удаление ключей из хэша

Другая неочевидная операция — удаление ключей из хэша. Как вы уже убедились, присвоение элементу хэша значения undef здесь не сработает. Для удаления одного ключа хэша можно воспользоваться функцией delete:

```
delete $Hash{keyval};
```

Для удаления всех ключей и значений из хэша можно просто инициализировать хэш пустым списком:

```
%Hash=();
```

Практическое применение хэшей

Хэши часто используются в Perl не только для хранения записей, доступ к которым выполняется по значению ключей. Основные преимущества хэшей — быстрый доступ к отдельному ключу и уникальность ключей. Эти особенности неоценимы при обработке данных. Из-за сходства с массивами хэши будут особенно полезны для преобразования массивов данных.

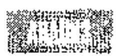
Определение частоты появления слов

На 6-м занятии, "Поиск по шаблону", вы узнали, как разделить строку текста на отдельные слова. Посмотрите на следующий фрагмент кода:

```
while(<>)  {
    while ( /(\w[\w-]*)/g  ) { t Итерация по всем словам с помощью переменной
$1.
      }    $Words{$1}++;
}
```



Проведем анализ этого кода. В первой строке выполняется чтение информации со стандартного устройства ввода. При этом введенные значения поочередно присваиваются переменной $_. Следующий цикл while совершает итерацию по всем словам, находящимся в переменной $_. Из 6-го занятия, "Поиск по шаблону", вы должны помнить, что оператор поиска по шаблону (//) в скалярном контексте и с модификатором g возвращает истинное значение до тех пор, пока не будут найдены все совпадения с шаблоном. Шаблон состоит из



текстового символа \w, за которым может следовать любое количество (включая нуль) текстовых символов или дефисов [\w-]*. Для того чтобы запомнить совпадение в специальной переменной $1, весь шаблон взят в скобки.

В следующей короткой строке кода происходят интересные вещи. Переменной $1 поочередно присваивается каждое слово, соответствующее шаблону из второй строки кода. Слова используются в качестве ключей хэша %Words. Первоначально значение пары ключ-значение не определено. Инкремент неопределенного значения, соответствующего впервые встреченному слову, помещает в элемент хэша значение 1. Если слово встречается во второй раз, ключ хэша уже существует и соответствующее ему значение увеличивается на 1, т.е. становится равным 2. Этот процесс продолжается, пока не закончатся входные данные.

После окончания работы кода в хэше %Words будет содержаться частота появления вводимых вами слов. Для того чтобы просмотреть полученное распределение, можно воспользоваться следующим кодом:

```
foreach( keys %Words) {
        print "$_ $Words{$_}\n";
```

Нахождение уникальных элементов массива

Описанную выше методику можно использовать для нахождения тех элементов массива, которые встречаются в нем только один раз. Предположим, вы извлекли все слова из введенных данных и поместили их не в хэш, а в массив. Допустим также, что проверка наличия в массиве данного слова перед его помещением туда не выполнялась. В этом случае в полученном списке могут содержаться повторяющиеся слова.

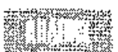
Предположим, что во вводимой строке будет содержаться следующий список слов:

```
@fishwords=('one', 'fish', 'two', 'fish', 'red', 'fish', 'blue', 'fish');
```

При помощи хэша данный список слов может быть проверен на наличие уникальных элементов, как показано в листинге 7.1.

Листинг 7.1. Нахождение уникальных элементов массива

```
1: %seen=();
2: foreach(@fishwords) {
3:     $seen{$_}=1;
4: }
5: @uniquewords=keys %seen;
```



Проведем анализ программы.

- *Строка 1.* В этой строке инициализируется временный хэш %seen, предназначенный для хранения слов.
- *Строка 2.* В этой строке совершается итерация по списку слов, переменной $_ поочередно присваиваются все слова.
- *Строка 3.* Здесь создается элемент хэша %seen, ключ которого хранится в переменной $_, а значение равно 1.

- *Строка 5.* Все ключи извлекаются из хэша и помешаются в массив @uniquewords. Все повторяющиеся слова, например fish, записываются в хэш поверх друг друга, поэтому в итоге они будут представлены там только одним ключом.

Вычисление пересечения и разности массивов

Часто возникает задача найти пересечение массивов (т.е. выделить общие элементы массивов) и разность массивов (т.е. выделить элементы, которые не встречаются сразу во всех массивах). В следующем примере один список содержит имена кинозвезд, а другой — имена политиков. Задача — найти политиков, одновременно являющихся кинозвездами. Вот два этих массива:

```
@stars=('R. Reagan', 'C. Eastwood', 'M. Jackson', 'Cher', 'S. Bono');
@pols = ('N. Gingrich', 'S. Thurmon', 'R. Reagan',
          'S. Bono', 'C. Eastwood', 'M. Thatcher');
```

Код для поиска пересечения этих массивов приведен в листинге 7.2.

Листинг 7.2. Нахождение пересечения массивов

```
1: %seen=();
2: foreach (@stars) {
3:     $seen{$_}=1;
4: }
5: @intersection=grep($seen{$_}, @pols);
```

 Проведем анализ программы.

- *Строка 1.* Инициализируется хэш %seen, предназначенный для хранения имен кинозвезд.
- *Строка 2.* В этой строке происходит итерация по списку кинозвезд с поочередным присваиванием каждого имени переменной $_.
- *Строка 3.* Здесь создаются новые элементы хэша %seen с именами кинозвезд в качестве ключей и значением 1, вместо которого можно использовать любое истинное значение.
- *Строка 5.* Эта строка выглядит сложнее, чем она есть на самом деле. Функция grep совершает итерацию по списку политиков @pols, поочередно присваивая имя каждого политика переменной $_. Затем проверяется наличие этого имени в хэше %seen. Если имя там существует, связанное с ним значение истинно. Если значение выражения истинно, $_ поступает в массив @intersection. Процесс продолжается до тех пор, пока grep не проверит каждый элемент массива gpols. После окончания выполнения этого кода в массиве @intersection будут содержаться имена, встречающиеся и в @stars, и в @pols.

Код для нахождения разности двух массивов, т.е. элементов, которые есть только в одном из массивов, практически идентичен предыдущему и приведен в листинге 7.3.

Листинг 7.3. Нахождение разности массивов

```
1: %seen=();
2: foreach (@stars) {
3:     $seen{$_}=1;
4: }
5: @difference=grep(! $seen{$_}, @pols);
```

Изменения коснулись лишь пятой строки программы. В ней по-прежнему проверяются все имена политиков на наличие в хэши `%seen`, но выражение функции `grep` получает ложное значение, если имя найдено, и истинное — если имя не найдено. Все имена политиков, содержащиеся в `%seen`, *не* возвращаются массиву `@difference`. Для нахождения всех кинозвезд, не являющихся политиками, используется почти тот же код, лишь меняются местами массивы `@stars` и `@pols`.

Сортировка хэшей

Часто ключи хэша нужно вывести в определенном порядке, а не в том, в котором они там хранятся. Скажем, вы хотите вывести частоту появления слов в алфавитном порядке или в порядке увеличения частоты. Так как функция keys возвращает список, к нему можно применить функцию sort для упорядочивания исходного списка:

```
foreach ( sort keys %Words ) {
      print "$_ $Words{$_}\n";
}
```

Сортировка по значениям частот ненамного сложнее. Как вы помните из 4-го занятия, "Укладка строительных блоков: списки и массивы", функция sort по умолчанию сортирует список согласно кодам ASCII. Если требуется более сложный вариант сортировки, функция sort вызывается вместе с блоком, определяющим порядок сортировки. Следующий код сортирует хэш по значениям:

```
foreach ( sort { $Words{$a} <=> $Words{$b} } keys %Words ) {
    print "$_ $Words{$_}\n";
}
```

Вы должны помнить, что блок функции sort вызывается многократно, причем переменные $a и $b в нем представляют каждую пару значений, которые должна сравнить sort. В нашем случае переменные $a и $b приобретают значения различных ключей хэша `%Words`. Вместо $a и $b сравниваются значения хэша `%Words`, соответствующие этим ключам.

Упражнение: создание в Perl простой базы данных пользователей

ЕСЛИ вам приходилось звонить в центр гарантийного обслуживания, вы, наверное, обратили внимание, что человек на другом конце провода вначале спрашивает ваш телефонный номер, номер гарантийного талона, номер карточки социального страхования или какой-нибудь другой номер, позволяющий компьютеру вас идентифицировать. Эти номера — не что иное, как ключи базы данных, позволяющие получить доступ к информации о вас. Не правда ли, все это похоже на ключи хэшей в Perl?

В этом упражнении мы выполним поиск в базе данных клиентов. Предполагается, что база данных уже создана, однако пока в ней не предусмотрены средства модификации записей. В нашем примере мы предоставим пользователю возможность поиска информации в двух различных полях.

Для того чтобы выполнить упражнение, нам нужны исходные данные. Запустите текстовый редактор, наберите приведенный ниже текст и сохраните его в файле customers.txt. Обратите внимание, что количество пробелов между столбцами не имеет особого значения. В большинстве случаев будет достаточно одного пробела.

```
Smith,John    (248)-555-9430 jsmitheaol.com
Hunter,Apryl  (810)-555-3029 april@showers.org
Stewart,Pat   (405)-555-8710 pats@starfleet.co.uk
Ching,Iris    (305)-555-0919 iching@zen.org
Doe,John      (212)-555-0912 jdoe@morgue.com
Jones,Tom     (312)-555-3321 tj2342iaol.com
Smith,John    (607)-555-0023 smith@pocahontas.com
Crosby,Dave   (405)-555-1516 cros@csny.org
Johns,Pam     (313)-555-6790 pjisleepy.com
Jeter,Linda   (810)-555-8761 netlessiearthlink.net
Garland,Judy  (305)-555-1231 ozgalirainbow.com
```

Наберите программу, приведенную в листинге 7.4, и сохраните ее в том же каталоге под именем Customer. Не забудьте сделать файл программы выполняемым, воспользовавшись инструкциями, приведенными на 1-м занятии, "Начало работы с Perl".

После этого попытайтесь запустить программу, набрав в командной строке

```
perl Customer
```

В листинге 7.5 содержится пример диалога с программой Customer.

Листинг 7.4. Исходный текст программы Customer

```
#!/usr/bin/perl -w

open(PH, "customers.txt") or die "Ошибка при открытии customers.txt: $!\n";
while(<PH>) {
        chomp;
        ($number, $email)=(split(/\s+/, $_))[1,2];
        $Email{$email}=$_;
        $Phone{$number}=$_;
}
close(PH);

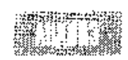
print "Для выхода из программы введите 'q'\n";
while (1) {
         print "\nНомер? ";
         $number=<STDIN>;  chomp($number);
         $address="";
        if (! $number ) {
                print "E-Mail? ";
                $address=<STDIN>;  chomp($address);
         >

        next if (! $number and ! $address);
        last if ($number eq 'q' or $address eq 'q'>;
```

```
24:
25:      if ( $number and exists $Phone{$number} ) {
26:             print "Заказчик: $Phone{$number}\n";
27:              next;
28:        }
29:
30:      if ($address and exists $Email{$address} } {
31:             print "Заказчик: $Email{$address}\n";
32:              next;
33:        }
34:       print "Заказчик не найден.\n";
35:        next;
36:}
37:print "\nПрограмма завершена.\n";
```

Листинг 7.5. Пример вывода программы Customer

```
1: Для выхода из программы введите 'q'
2: Номер? <Enter>
3: E-Mail? cros@csny.org
4: Заказчик: Crosby, Dave     (405)-555-1516      cros@csny.org
5:
6: Номер? (305)-555-0919
7: Заказчик: Ching, Iris    (305)-555-0919 iching@zen.org
8:
9: Номер? q
10:
11:Программа завершена.
```

 Проведем анализ программы.

- *Строка 1.* В этой строке указывается путь к интерпретатору (измените его в соответствии с конфигурацией вашей системы) и ключ -w. Всегда включайте режим выдачи предупреждений!
- *Строка 3.* Открывается файл customers.txt и ему назначается дескриптор PH. Естественно, проверяется успешность выполнения этой операции и при необходимости выводится сообщение об ошибке.
- *Строки 4S.* В цикле выполняется чтение строк из дескриптора PH (каждая строка по очереди присваивается переменной $_). Символ перевода строки, находящийся в переменной $_, удаляется с помощью функции chomp.
- *Строка 6.* Информация, находящаяся в переменной $_, разделяется по словам с помощью шаблона /\s+/, соответствующего одному или нескольким пробелам. Выражение с функцией split взято в скобки и за ним следуют две цифры в квадратных скобках. Так как нас интересуют телефонный номер и адрес электронной почты, то используются только указанные части списка, производимого функцией split, и переменным $number и $email присваиваются соответствующие значения.

- *Строки 7-8.* Для доступа к записям о клиентах по адресу электронной почты и по номеру телефона используются два хэша: %Email и %Phone. В первом хэше в качестве ключей используются адреса электронной почты, а во втором — номера телефонов.
- *Строка 10.* В этой строке закрывается дескриптор рн.
- *Строка 13.* Организовывается цикл while, выполняющий повтор блока кода. Выражение while (1) — идиома Perl для организации бесконечного цикла. Для выхода из этого цикла служит оператор last.
- *Строки 14–15.* Считывается номер телефона и удаляется символ перевода строки.
- *Строки 17–20.* Если номер телефона не введен, предлагается ввести адрес электронной почты.
- *Строки 22–23.* Если ничего не введено, цикл повторяется. Если пользователь вводит символ q, происходит выход из цикла.
- *Строки 25–28.* Если введен допустимый телефонный номер, строка 26 кода выводит запись о клиенте. Управление передается в начало блока с помощью оператора next.
- *Строки 30–33.* Если введен допустимый адрес, выводится запись о клиенте. Управление передается в начало блока с помощью оператора next.

В данном примере продемонстрировано сразу несколько возможностей Perl. Хэши используются для быстрого доступа к данным посредством ключей. В Perl хэши реализованы весьма эффективно, время ответа будет приемлемым, даже если в базе данных содержатся тысячи или десятки тысяч записей. Кроме того, на примере этой программы показано, как можно организовать управление процессом выполнения программы в простом блоке без использования управляющих структур while, do, until и им подобным.

Резюме

Хэши предоставляют программисту Perl множество новых возможностей. Помимо простого хранения и извлечения записей, хэши используются для обработки и анализа данных. Методы обработки массивов, хранения и получения доступа к записям имеют огромное значение. В следующих занятиях знание хэшей облегчит вам изучение таких тем, как работа с DBM-файлами, сложные структуры данных, взаимодействие с системным окружением.

Вопросы и ответы

Можно ли хранить в элементе хэша больше одного значения, т.е. список, адресуемый одним ключом?

Да. Для этого существует два основных способа. Первый (и довольно неудобный) состоит в форматировании значения элемента в виде обычной записи, например списка с элементами, разделенными запятыми. Для преобразования списка в хэш используется функция join, а при извлечении значения из хэша скаляр разделяется функцией split. Этот метод неудобен и его реализация часто сопровождается ошибками.

Второй способ — использование *указателей.* Указатели позволяют создавать хэши массивов, хэши хэшей и другие сложные структуры данных. Если вы разберетесь с указателями, вам не составит особого труда создать сложные структуры. Эта тема подробно рассматривается на 13-м занятии, "Структуры и ссылки".

Как можно сохранить ключи в том порядке, в котором они были введены?

Это можно сделать несколькими способами. Первый способ подразумевает, что вы сами должны следить за порядком ввода элементов. Для этого можно использовать специальный массив. При вводе нового элемента в хэш в массив вводится ключ этого элемента с помощью функции `push`. Если вам потребуется при доступе к хэшу восстановить порядок ввода записей, используйте этот массив, а не функцию keys. Реализовать без ошибок этот метод очень сложно.

Гораздо лучший способ — использовать модуль Tie: :IxHash, который заставляет функцию keys выводить ключи в порядке ввода их элементов, что вам было и нужно. Подробнее об использовании модулей речь пойдет на 14-м занятии, "Использование модулей".

Существует ли удобный способ записи хэша в файл?

Конечно. Модули `Data::Dumper` и `Storable` могут преобразовать типы данных в скалярные значения, которые можно легко сохранить в текстовом файле. В этих модулях есть функции для обратного преобразования этих скаляров в исходные структуры данных.

На 15-м занятии, "Обработка данных в Perl" будет рассмотрен еще более простой способ записи хэша в файл с использованием DBM-файлов. DBM-файлы позволяют связать хэш с файлом на диске. При изменении хэша будет изменяться и содержимое файла. Таким образом, с помощью файлов на диске можно организовать длительное хранение содержимого хэша.

Семинар

Контрольные вопросы

1. Почему в программе Customer нельзя в качестве ключей хэшей использовать имена людей?

   а) в элемент хэша Perl невозможно одновременно поместить и имя, и фамилию клиента;

   б) имена людей не могут быть уникальными ключами;

   в) имена не используются для поиска в базе данных.

2. Какое различие существует между ассоциативным массивом и хэшем?

   а) нет никакого различия;

   б) ассоциативные массивы используются для хранения более формализованных наборов данных, таких как записи бухгалтерской системы;

   в) хэши в Perl — это не совсем ассоциативные массивы, поэтому и термин другой.

3. Какой вид данных больше подходит для хэшей?

   а) простой список элементов;

   б) массив;

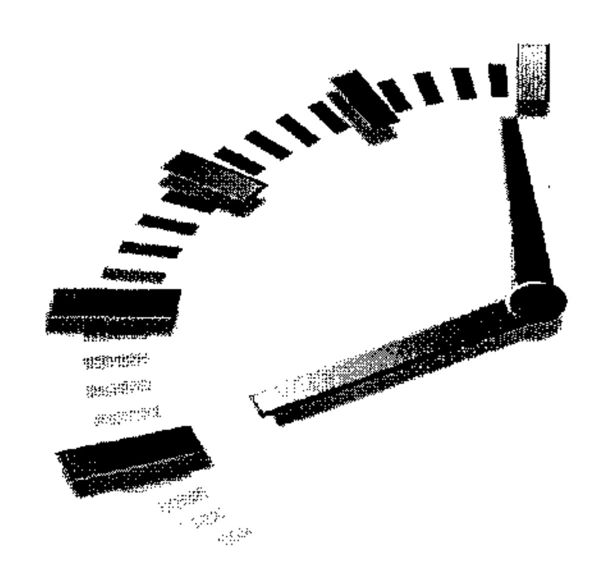
   в) список пар ключ-значение.

Ответы

1. Правильный ответ — вариант б). Размер хэша Perl не ограничен, а клиенты часто просят найти их запись по имени. Однако имена людей не подходят, потому что они не уникальны. В телефонных книгах очень много одинаковых имен, таких как John Smith или Robert Jones.
2. Правильный ответ — вариант а). Хэши и ассоциативные массивы — одно и то же. Единственная разница между ними — тот факт, что слово *хэш* короче.
3. Правильный ответ в), хотя для инициализации хэши можно использовать и массив, преобразованный в список.

Упражнения

- Модифицируйте программу Customer таким образом, чтобы можно было получать информацию по имени клиента. Поскольку в качестве ключей нельзя использовать имена клиентов, вам придется организовать поиск нужного значения элемента хэша.
- Модифицируйте программу Customer так, чтобы для поиска клиентов можно было использовать часть ключа (например, часть телефонного номера или электронного адреса). Для этого воспользуйтесь регулярными выражениями, шаблоном в которых служит часть ключа. Имейте в виду, что может быть найдено сразу несколько записей и все они должны быть выведены.



8-й час

Функции

Почти все языки программирования содержат *функции*. Функция — это фрагмент кода, вызываемый по имени и возвращающий некоторое значение. В этой книге вы уже встречались с функциями print, reverse, sort, open, close, split и др. Но то были встроенные функции Perl.

Perl предоставляет вам возможность написания собственных функций. Определенные пользователем функции называются в Perl *подпрограммами*. Как и встроенные функции, они вызываются по имени и возвращают значение.

В Perl также реализована концепция *области видимости*. Область видимости определяет набор переменных, доступных программе в определенный момент времени. Благодаря этой концепции можно создавать функции, полностью независимые от остальной части программы. Корректно написанные функции могут быть повторно использованы и в других программах.

Основные темы этого занятия.

- Определение собственной функции и ее вызов.
- Передача значений в функции и получение возвращаемых ими значений.
- Использование директивы use strict для ужесточения контроля за кодом.

Создание и вызов подпрограмм

Для создания пользовательских подпрограмм в Perl используется следующий синтаксис:

```
sub имя_подпрограммы {
        оператор1;
        ...
        операторN;
>
```

Имена подпрограмм подчиняются тем же соглашениям об именах, что и имена скаляров, массивов и хэшей. Эти соглашения рассматривались на 2-м занятии, "Строительные блоки Perl: числа и строки". Имена подпрограмм могут совпадать с именами переменных.

Но вам следует избегать создания подпрограмм с именами, как у встроенных функций и операторов Perl. Создание двух подпрограмм с одинаковыми именами приводит к появлению соответствующего предупреждения, если включен режим вывода предупреждений, в противном случае второе определение функции перекрывает первое.

При вызове подпрограммы запускается ее код и возвращаемое значение передается в то место, из которого была вызвана подпрограмма. Например, данная короткая подпрограмма используется для диалога с пользователем:

```
sub yesno {
      print "Вы в этом уверены (Да/Нет)?";
      $answer=<STDIN>;
}
```

Для инициирования подпрограммы, или, как еще говорят, ее вызова можно использовать подобные строки:

```
&yesno();
```

или

```
yesno();
```

Второй синтаксис (без &) можно использовать, если подпрограмма была ранее определена в коде, а синтаксис &yesno( )допустим в любом месте кода. В этой книге используются вызовы подпрограмм без S, хотя обе формы синтаксиса могут иметь место.

При вызове подпрограммы Perl запоминает место, в котором произошел вызов, выполняет код подпрограммы и затем возвращается в основную программу к месту вызова:

```
sub countdown {
      for($i=10; $i>=0; $i--) {
          print "$i -";
      }
}
print "Обратный отсчет: ";
countdown();
print "Пуск!";
```

Подпрограммы Perl могут быть вызваны в любом месте основной программы и даже из других подпрограмм:

```
sub world {
      print "World!";
}
sub hello {
      print "Hello, ";
      world();
}
hello();
```

Возврат значений из подпрограмм

Подпрограмма служит не только для удобной группировки кода с возможностью его вызова по имени. Подпрограмма, как и функции, операторы и выражения Perl, имеет значение. Оно называется *возвращаемым значением.* Возвращаемое значение подпрограммы — значение последнего вычисленного в подпрограмме выражения или же значение, явно возвращаемое оператором return.

Значение подпрограммы вычисляется после вызова подпрограммы и затем используется в том месте, из которого подпрограмма была вызвана. Вот пример:

```
sub two_by_four {   # Простейшая подпрограмма
    2*4;
}
print 8*two_by_four();
```

Здесь Perl при вычислении выражения 8*two_by_four{) вызывает подпрограмму two_by_four(), которая возвращает значение 8. Затем вычисляется выражение 8*8 и в итоге выводится 64.

Значения подпрограмм также могут явно возвращаться оператором return. Этот оператор используется для досрочного выхода из кода подпрограммы и явного указания возвращаемого значения. В приведенном ниже примере используются оба способа выхода:

```
subx_greaterthan100 {
    f Используется установленное ранее значение $x
    return (1) if ($x>100);
    0;
}
$x=70;
if (x_greaterthan100()) {
   print "$x is greater than 100\n";
}
```

Подпрограммы могут возвращать не только скаляры, а и массивы или хэши:

```
sub shift_to_uppercase {
   @words=qw( cia fbi un nato unicef );
   foreach (@words) {
     $_=uc($J;
   }
   return(@words);
}
@acronyms=shift_to_uppercase(J
```

Аргументы

У всех предыдущих примеров общим было то, что данные, с которыми они работали, были или литералами (2*4), или переменными главной программы ($x в x_greaterthan100). Эти ограничения создают проблему переносимости кода подпрограмм. Для преодоления этих ограничений нужно уметь вызвать функцию и сказать ей: "Возьми *эти* данные и выполни с ними некоторые действия", а затем сказать ей: "Возьми *другие* данные и выполни с ними некоторые действия". Поведение функции должно изменяться в зависимости от передаваемых ей данных.

Значения, передаваемые функции и изменяющие ее поведение, называются *аргументами* (мы уже неоднократно использовали в книге этот термин). Вы уже знаете, что встроенным функциям Perl (grep, sort, reverse, print и др.) можно передавать аргументы. То же относится и к вашим подпрограммам. Для передачи функции аргументов можно использовать любой синтаксис:

```
имя_функции(арг1, арг2, арг3);
имя_функции  арг1, арг2, арг3;
& имя_ функции(арг1,арг2, арг3);
```

Вторая форма — без скобок — может быть использована, только если интерпретатор Perl уже встречал определение этой функции.

В подпрограмме доступ к аргументам осуществляется посредством специальной переменной @_. В следующем примере демонстрируется передача трех строковых литералов в подпрограмму, которая выполняет их печать в виде списка.

```
sub printargs {
    print join(',', @_);
}
printargs ('market', 'home', 'roast beef');
```

Для доступа к индивидуальным аргументам можно использовать индексы массива @_, как и в случае любого другого массива. Однако следует помнить, что конструкция $_[0] не имеет никакого отношения к скалярной переменной $_.

```
sub print_third_argument {
    print $_[2];
}
```

Использование в качестве имен переменных конструкции типа $_[3] — не очень прозрачный стиль программирования. Функции, в которых много переменных, как правило, начинаются с переименования этих переменных ~ после этого становится понятно, для чего они предназначены. Посмотрите, что имеется в виду:

```
sub display_box_score {
     ($hits, $at_bats)=@_;
     print "Из $at_bats попыток было выбито $hits очков.";
     print "Общий результат ~ ", $hits/$at_bats, "\n";
}
display_box_score(50, 210);
```

Здесь массив @_ копируется в список ($hits, $at_bats). Первый элемент массива @_ — @_[0] — становится переменной ?hits, а второй — переменной $at_bats. Имена переменных использованы лишь для читабельности.



> Переменная @_ содержит реальный список исходных аргументов подпрограммы. Ее изменение или изменение любого ее элемента приводит к модификации соответствующих переменных. Неявное изменение значения переменных — плохой стиль программирования, поэтому ваша функция не должна изменять значения передаваемых ей аргументов, если только это не делается осознанно.

Передача массивов и хэшей

Аргументы подпрограмм не обязательно должны быть скалярами. Вы можете передавать в подпрограмму массив ли хэш, но делать это нужно с умом. Массив ли хэш передается подпрограмме так же, как и скаляр:

```
@sorted_itens=sort_numerically(@items);
```

В этой подпрограмме обращение к массиву @items осуществляется посредством переменной @_:

```
sub sort_numerically {
    print "Сортировка...";
    return(sort {$a <=> $b} @_);
}
```

При передаче подпрограмме массива или хэша вы рискуете здорово ошибиться, если таких массивов несколько. Посмотрите на данный фрагмент кода:

```
sub display_arrays {
    (@a, @b)=@_;
     print "Первый массив: @a\n";
     print "Второй массив: @b\n";
}
dispay_arrays(@first, isecond);
```

Массивы @first и @second передаются при вызове подпрограммы в одном списке @_. Внутри этого большого плоского списка невозможно определить конец первого массива и начало второго. В подпрограмме оператор присваивания (@a, @b)=@_ приводит к тому, что все элементы массива @_ становятся элементами массива @a, а массив @b так и останется пустым списком. Чтобы вспомнить, почему это происходит, обратитесь к материалу 4-го занятия, "Укладка строительных блоков: списки и массивы".

Можно довольно просто передать в подпрограмму группу скалярных переменных и один массив или хэш. Для этого достаточно знать количество передаваемых скаляров и поместить массив или хэш последним в списке аргументов. В этом случае понятно, что после скаляров все оставшиеся аргументы принадлежат массиву или хэшу:

```
sub lots_of_args {
    ($first, $second, $third, @array)=@_;
     # Остальная часть подпрограммы...
}
lots_of_args($foo, $bar, $baz, @myarray);
```

Если нужно передать в подпрограмму несколько массивов или хэшей, а затем отделить их друг от друга, используйте *указатели.* О том, как передавать в подпрограммы указатели, речь пойдет на 13-м занятии, "Структуры и ссылки".

Область видимости

Из введения к этому занятию вы узнали, что подпрограммы позволяют сгруппировать фрагменты кода и дать им имена, которые затем можно использовать для вызова этих фрагментов кода из любого нужного места программы. Подпрограммы позволяют создать функционально законченные модули, решающие универсальные задачи. На основе переданных аргументов, встроенных функций и выражений подпрограмма возвращает определенное значение. Вы можете использовать эту подпрограмму в других программах, так как ее выполнение не зависит от окружающего контекста, — она просто берет аргументы, внутренние данные и на их основании вычисляет возвращаемое значение. Функция становится как бы "черным ящиком", который связывают с внешним миром только входные и выходные значения. Такие функции называются *строгими.*

А теперь давайте проанализируем два следующих фрагмента кода.

```
t Один из хороших стилей написания функции
sub moonweight {
    ($weight)=@_;
     return ($weight/6);
}
print moonweight(150);

# Плохой стиль написания подобной функции.
```

```
sub moonweight {
    return($weight/6);
}
$weight=150;
print moonweight;
```

По большому счету, первая реализация функции намного лучше. Она не использует предустановленных внешних переменных. Вместо этого она копирует аргумент в переменную $veight и производит все вычисления с ней. Вторую реализацию трудно использовать в другой программе, для этого нужно, чтобы переменной **$weight** было присвоено соответствующее значение и обозначало то, что нам нужно. Если переменная с таким именем уже используется в других целях, вам придется изменить имя переменной, используемое в функции **moonweight**(), что не очень удобно.

Итак, первый вариант реализации лучше, но в нем по-прежнему чего-то не хватает. Переменная $weight может конфликтовать с другой переменной с подобным именем, если та встречается в программе.

Perl позволяет использовать одни и те же имена для обозначения различных переменных в большой программе. По умолчанию переменные Perl видимы в основной программе и в подпрограммах. Это так называемые *глобальные переменные.*

Допустим, вам нужно создать переменную, относящуюся только к данной функции. Для этого следует воспользоваться оператором **my**:

```
sub moonweight {
    my $weight;
    ($weight)=@_;
    return($weight/1.66667);
}
```

Внутри функции moonweight() переменная $weight является *приватной переменной.* Другие функции программы не имеют доступа к этой переменной. Таким образом, все другие переменные с именем $weight не пересекаются с переменной $weight функции **moonweight()**. Теперь подпрограмма совершенно изолирована от остальной части программы.

Та часть программы, в которой можно использовать данную переменную, называется *областью видимости* переменной.

Оператор **my** можно использовать для объявления скаляров, массивов и хэшей, как приватных переменных подпрограммы. Дескрипторы файлов, подпрограммы и специальные переменные Perl (**$!**, **$_**, **@_** и др.) нельзя объявить приватными переменными. Можно объявлять сразу несколько переменных, поместив их имена в скобки:

```
my ($larry, @curly, %moe);
```

Приватные переменные хранятся совсем не так, как глобальные. Глобальные и приватные переменные могут иметь одинаковые имена, но при этом не конфликтовать, как показано ниже:

```
sub myfunc {
    my $x;
    $x=20;   \ Это приватная переменная $x
    print "$x\п";
)
$x=10;   i Это глобальная переменная $x
print "$x\n";
myfunc();
print "$x\n";
```

В результате выполнения этого примера будут выведены числа 10, 20, а затем снова 10. Переменные $x в функции myfunc() и $x в основной части программы — совершенно разные. У вас может возникнуть один резонный вопрос: "Возможно ли в подпрограмме использовать как приватную, так и глобальную переменную с одним и тем же именем?" Да, но это достаточно сложный вопрос, обычно не рассматриваемый начальными учебными пособиями по Perl.

Как правило, любая подпрограмма на Perl начинается с объявления приватных переменных и присваивания массива @_ списку этих переменных:

```
sub playerstats {
    my ($at_bats, $hits, $walks) =@_;
    # Остальная часть функции...
}
```

Эта техника программирования позволяет создавать удобные в использовании, или, как их еще называют, дружественные функции: все их переменные являются приватными, поэтому эти переменные не могут изменить или быть изменены другими функциями, или основной частью программы. После окончания подпрограммы ее приватные переменные аннулируются.

Использование оператора my

Можно объявить переменные даже с меньшей областью видимости, чем целая подпрограмма. Для этого следует поместить оператор my в блок. Это может быть как основной блок подпрограммы, так и какой-нибудь другой блок. В следующем примере приватная переменная видима только в блоке:

```
$y=20;
{
     my $y=500;
     print "Значение переменной \$y равно $y\n";
     # Напечатает 500
}
print "$y\n"; # Напечатает 20
```

Объявлять переменные можно даже внутри управляющей структуры, такой как for, foreach, while или if. В общем, какой бы у вас ни был блок, можно создать переменную с областью видимости внутри этого блока, как показано в следующем примере:

```
while($testval) {
    my $stuff;  # Переменная видима только внутри цикла
    ...
}
foreach(@t)  {
    my %hash;  # Хэш видим только внутри цикла foreach
}
```

Здесь переменные, объявленные в операторе my, заново создаются во время каждой итерации цикла.

В Perl версии 5.004 и выше можно переменную (итератор) цикла for или foreach, а также условное выражение структур while или if объявить приватными для блока:

```
foreach my $element (@array) {
# Переменная $element видима только внутри цикла foreach()
}
```

```
while(my $line=<STDIN>) {
# Переменная $line видима только внутри цикла while()
}
```

По окончании блока его приватные переменные аннулируются.

Упражнение: подсчет статистики

После того как вы столько узнали о функциях, перейдем к практическому использованию функций в реальном приложении. Код функций легко может быть использован повторно. В этом упражнении три функции предназначены для анализа групп цифр.

Для начала немного теории. Как вы должны помнить из курса средней школы, *среднее арифметическое* группы цифр — это их сумма, деленная на количество цифр. *Медиана* — это средний элемент всей группы, отсортированной по значению. При четном количестве элементов берется среднее арифметическое двух средних элементов. *Стандартное отклонение* характеризует распределение элементов вокруг среднего значения. Высокое значение стандартного отклонения означает, что разброс чисел велик, а малое — что они сосредоточены вокруг среднего значения, Если отложить влево и вправо от среднего значения интервал, равный стандартному отклонению, то в нем будет сосредоточено 68% чисел из набора, а если удвоить этот интервал, то в него попадет 95% всех чисел из набора.

А теперь, вооружившись теоретическими сведениями, наберите в текстовом редакторе программу, приведенную в листинге 8.1, и сохраните ее под именем Stats. Не забудьте сделать файл выполняемым, воспользовавшись инструкциями из 1-го занятия, "Начало работы с Perl".

После этого попытайтесь запустить программу, набрав в командной строке

```
perl Stats
```

В листинге 8.2 содержится пример диалога с этой программой.

Листинг 8.1. Исходный текст программы Stats

```
#!/usr/bin/perl -w

use strict;
sub mean {
      my(@data)=@_;
      my $sum;
      foreach(@data) {
            $sum+=$_;
      }
      return($sum/@data);
}
sub median {
    my(@data)=sort { $a <=> $b} @_;
    if (scalar(@data)%2) {
            return($data[@data/2]);
    } else {
            my($upper, $lower);
             $lower=$data[@data/2];
            $upper=$data[@data/2 - 1];
            return(mean($lower, $upper)};
```

```
21:         }
22: }
23: sub std_dev {
24:         my(@data)=@_;
25:         my($sq_dev_sum, $avg)=(0,0);
26:
27:         $avg=mean(@data);
28:         foreach my $elem (@data) {
29:                 $sq_dev_sum+=($avg-$elem)**2;
30:         }
31:         return(sqrt($sq_dev_sum/(@data-1)));
32: }
33: my($data, @dataset);
34: print "Введите данные, разделенные запятыми: ";
35: $data=<STDIN>; chomp $data;
36: @dataset=split(/[\s,]+/, $data);
37:
38: print "Медиана: ", median(@dataset), "\n";
39: print "Среднее: ", mean(@dataset), "\n";
40: print "Станд. откл.: ", std_dev(@dataset), "\n";
```

Листинг 8.2. Пример диалога с программой Stats

```
Введите данные, разделенные запятыми: 14.5,6,8,9,10,34
Медиана: 9.5
Среднее: 13.5833333333333
Станд. откл.: 10.3943093405318
```

 Проведем анализ программы.

- *Строка 1.* В этой строке указывается путь к интерпретатору (измените его в соответствии с конфигурацией вашей системы) и ключ -w. Всегда включайте режим вывода предупреждений!
- *Строка 3.* Директива use strict указывает, что все переменные должны быть явно объявлены в программе с помощью оператора my и строки должны быть заключены в кавычки.
- *Строки 4—11.* Функция mean() находит среднее арифметическое группы чисел. С помощью цикла foreach вычисляется сумма чисел $sum, а затем она делится на количество чисел.
- *Строки 12—21.* Функция median() вычисляет данные двумя способами. При нечетном количестве вводимых элементов она просто выбирает средний элемент. Для этого количество элементов делится на 2, получившееся значение округляется до ближайшего целого и используется в качестве индекса предварительно отсортированного массива с нашими числами. При четном количестве элементов функция находит два средних элемента. Эти числа — $upper и $lower. Их среднее находится с помощью функции mean(). Это значение и возвращается функцией вычисления медиан.

- *Строки 23—32.* Функция std_dev() очень проста и состоит в основном из математического выражения. В этом выражении из каждого элемента массива §data вычитается среднее значение всех элементов и получившееся число возводится в квадрат. Все подобные результаты суммируются в переменной $sq_dev_sum. Для нахождения стандартного отклонения сумма всех квадратичных отклонений делится на количество элементов минус единица, и из получившегося значения берется квадратный корень.
- *Строки 33—35.* Все необходимые переменные основной программы объявляются с помощью оператора my. Пользователь вводит данные, которые помещаются в скаляр Sdata. Затем с помощью функции split и шаблона /[\s,]+/ введенные пользователем данные разделяются и помещаются в массив @dataset. Этот шаблон определяет в качестве разделителя символы пробела и запятую. Дополнительные пробелы и запятые игнорируются.
- *Строки 38—40.* Генерируется вывод. Не забывайте, что это не единственное место, где можно вызвать функции mean(), median() и std_dev(). Они могут вызывать друг друга: обе функции std_dev() и median{} используют функцию mean(). Это неплохой пример повторного использования кода.

Подробнее о функциях

Многие технические приемы программирования, как, например, рекурсивный вызов функций, могут быть эффективно реализованы только с учетом области видимости переменных. К этому вопросу непосредственное отношение имеет оператор use strict, ужесточающий синтаксический контроль в Perl, что позволяет избежать некоторых грубых ошибок.

Объявление переменных с помощью оператора local

В Perl версии 4 не было по-настоящему приватных переменных. Вместо них в Perl 4 были "почти" приватные переменные. Концепция "почти" приватных переменных до сих пор присутствует и в Perl 5. Мы будем называть такие переменные локальными. Для объявления этих переменных используется оператор local:

```
sub myfunc {
   local ($foo)=56;
   # остальная часть функции...
>
```

В этом примере объявляется локальная переменная $foo подпрограммы myfunc(). Поведение переменной, объявленной с помощью оператора local, почти не отличается от поведения переменной, объявленной с помощью оператора my. Областью видимости локальной переменной может быть подпрограмма, блок или конструкция eval. При завершении подпрограммы или блока эта переменная аннулируется.

Отличие состоит в том, что переменная, объявленная локальной, видима не только в пределах того блока, в котором она объявлена, но и во всех подпрограммах, вызванных из этого блока. В табл. 8.1 приведено сравнение приватных и локальных переменных.

Таблица 8.1. Сравнение операторов и

```
sub mess_with_foo {
    $foo=0;
>
sub myfunc {
    my $foo=20;
    mess_with_foo();
    print $foo;
}
myfunc();
```

```
sub mess_with_foo {
    $foo=0;
}
sub myfunc {
    local $foo=20;
    mess_with_foo();
    print $foo;
}
myfunc();
```

Два фрагмента текста, приведенные в табл. 8.1, практически идентичны, за исключением объявления переменной $foo в функции `myfunc()`. Слева эта переменная объявлена как приватная, а справа — как локальная.

В коде слева функция `mess_with_foo()` изменяет *глобальную* переменную $foo. После возвращения управления в `myfunc()` будет напечатано число 20, так как приватная переменная $foo, принадлежащая этой функции, не была изменена.

В коде справа в функции `mess_with_foo()` создается *локальная* переменная $foo. Затем функция `mess_with_foo()` устанавливает для переменной $foo значение 0. Это та же переменная, что и в myfunc(), — локальные переменные передаются в подпрограммы. После возвращения в `myfunc()` будет напечатано значение 0.



Специально для любителей терминологии сообщим, что локальные переменные также называются переменными с *динамической областью видимости,* так как их область видимости изменяется по мере вызова подпрограмм. Переменная, объявленная с помощью оператора `my`, имеет *лексическую область видимости,* которая сравнительно просто определяется после анализа кода. Лексическая область видимости совпадает с блоком, в котором была объявлена данная переменная, причем s процессе выполнения программы эта область не изменяется.

Итак, если вам нужно объявить по-настоящему приватную переменную, используйте оператор ray.

Как сделать Perl строже

Perl — снисходительный и терпимый язык программирования. Он не обращает особого внимания на то, как выглядит ваш код, лишь бы он работал. Но имеется возможность указать интерпретатору Perl, чтобы тот более взыскательно анализировал исходный код. Например, предупреждения, включенные в командную строку или в первую строку программы, позволяют избежать многих досадных ошибок. Perl предупреждает об использовании неопределенных переменных, однократном использовании имен и подобных вещах.

При разработке больших программных проектов (особенно по мере усложнения программы) было бы неплохо, чтобы Perl хоть как-то удерживал вас от совершения случайных ошибок. Кроме ключа `-w`, в интерпретаторе Perl имеются и другие средства для вывода дополнительных предупреждений во время компиляции. Для этого используется оператор `use strict`:

```
use strict;
sub musub {
    my $x;
    :
}
mysub();
```

Оператор use strict называется *директивой компилятора.* Эта директива указывает Perl, что необходимо отслеживать перечисленные ниже ситуации и выводить сообщения об ошибках времени выполнения с информацией о текущем файле и блоке.

- Использование имени переменной, не являющейся специальной, без ее объявления в операторе my.
- Использование недопустимого имени функции (без & и скобок) перед тем, как эта функция была определена.
- Другие потенциальные ошибки

Директива use strict помогает справиться с первыми двумя проблемами. Теперь в программе вы уже не сможете использовать глобальную переменную вместо приватной. Это способствует созданию более изолированного кода, не полагающегося на использование глобальных переменных.

Другая ловушка, выпутаться из которой помогает директива use strict, — использование недопустимых ключевых слов. Посмотрите на код

```
$var=value;
```

В данном случае непонятно, что такое value — строка без кавычек или имя функции. При использовании директивы use strict Perl сообщит, что этот код непонятен и синтаксически недопустим, если только соответствующая функция предварительно не определена.

С этого момента все примеры и относительно длинные листинги в книге будут содержать директиву use strict.

Рекурсия

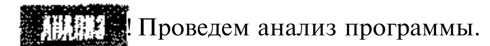
Рано или поздно вы познакомитесь со специальным классом подпрограмм. Во время своей работы эти подпрограммы вызывают сами себя. Они называются *рекурсивнымиподпрограммами.*

Рекурсивные подпрограммы используются в случае, если задача может быть сведена к решению более простой идентичной задачи, а та, в свою очередь, — к еще более простой, и т.д. Одна из рекурсивных задач — поиск файла в древовидной структуре каталогов: вначале поиск ведется в самом верхнем каталоге, затем наступает очередь подкаталогов и т.д. Как видите, в данном случае подзадачи идентичны основной задаче.

Другая рекурсивная задача связана с вычислением факториалов. Факториалы используются в статистике. Количество перестановок букв *ABCDEF* равняется факториалу шести. *Факториал* — это произведение всех целых чисел, меньших данного, включая 1. Например, факториал числа 6 — 6x5x4x3x2x1 или 720. Для вычисления факториала 6 необходимо факториал 5 умножить на 6. Для вычисления факториала 5 Необходимо факториал 4 умножить на 5 и т.д. Рекурсивная функция для нахождения факториалов приведена в листинге 8.3.

Листинг 8.3. Вычисление факториалов с помощью рекурсии

```
sub factorial {
   my ($num)=@_;
   return(1) if ($num <= 1);
    return($num*factorial($num-1));
}
print factorial(6);
```


Проведем анализ программы.

- *Строка 2.* Аргумент подпрограммы factorial!) присваивается переменной **$num**, которая объявлена в данной подпрограмме как приватная.
- *Строка 3.* Для каждой рекурсивной функции необходимо предусмотреть *условие прекращения.* Это то значение аргумента, при котором функция больше не вызывает сама себя. Для подпрограммы **factorial**() условие прекращения — это вычисление факториала 1 или 0. Два приведенных значения равны единице, поэтому подпрограмма factorial(), вызванная с аргументом 0 или 1, выполняет код **return(1)** (это случай $пшп<=1).
- *Строка 4.* Если аргумент не равен нулю или единице, вычисляется факториал предыдущего числа, как показано ниже.

Если $num равна...	***В строке 4 вычисляется...***
6	`return(6*factorial(5))`
5	`return(5*factorial(4))`
4	`return(4*factorial(3))`
3	return(3*factorial(3)}
2	`return(2*factorial(1))`
1	строка 4 не выполняется; функция factorial(l) возвращает значение 1

В результате выполнения цепочки вызовов рекурсивных функций будут последовательно вычислены факториалы всех чисел, меньших данного. Их значения по цепочке передаются в вызвавшие их функции, и в конце концов вычисляется значение факториала 6.

Рекурсивные функции встречаются не часто. Большие рекурсивные функции тяжело создавать и отлаживать. Любая задача, решаемая с помощью циклов for, while, foreach, может быть решена при помощи рекурсии, и, наоборот, всякая рекурсивная задача может быть выполнена с помощью циклов. Рекурсивные процедуры обычно используются для решения относительно небольшого круга задач, естественным образом формализуемых с помощью метода рекурсии.

Резюме

Perl поддерживает определяемые пользователем функции, называемые подпрограммами, которые ведут себя подобно встроенным функциям. Им можно передавать аргументы; при необходимости подпрограммы пользователя могут возвращать значения в место вызова. Функции Perl могут вызывать другие функции и даже самих себя. В Perl также можно объявить приватную переменную внутри функции или блока кода и создать законченные фрагменты кода, допускающие повторное использование.

Вопросы и ответы

Есть ли на самом деле какая-то разница при вызове функции с символом & и без него?

Нет ничего такого, что могло бы иметь к вам какое-нибудь отношение. Существует небольшая разница между вызовами *&foo* и foo при использовании прототипов функций

или отсутствии скобок после вызова. Но эта тема выходит за рамки нашей книги. Вы можете удовлетворить свое любопытство, обратившись к странице **perlsub** справочного руководства

В моей программе есть строка my($var), для которой Perl выводит сообщение об ошибке syntax error, next 2 tokens шу(.

Вы или что-то неправильно набрали, или у вас установлен Perl версии 4. Наберите в командной строке perl -v. Если у вас действительно четвертая версия, немедленно установите более новую.

Как передавать в подпрограмму или возвращать из нее функции, дескрипторы файлов, а также сразу несколько массивов или хэшей?

Для передачи функций, нескольких массивов или хэшей нужно использовать указатели, рассматриваемые на 13-м занятии, "Структуры и ссылки". Для передачи дескрипторов файлов в подпрограмму и обратно следует воспользоваться модулем IO::Handle или методикой приведения типов (typeglob). Обе эти темы выходят за рамки данной книги.

Моя функция возвращает много значений, а мне нужно только одно. Как пропустить остальные значения?

Один из методов — создание списка литералов, для чего вызов функции помешают в круглые скобки. Затем вы можете выбрать любую интересующую вас часть этого списка. В следующем примере извлекается значение года (текущий год минус 1900) из встроенной функции localtime, возвращающей сразу девять значений:

```
print "На дворе ", 1900+ (localtime)[5];
```

Другой метод — присваивание переменных списку таким образом, чтобы все ненужные значения присваивались undef или какой-нибудь вспомогательной переменной:

```
(undef, undef, undef, undef, undef, $year_offset)=localtime;
```

Семинар

Контрольные вопросы

Посмотрите на следующий блок кода:

```
sub bar {
    <$a, $b)=@_;
    $b=100;
    $a=$a+1;
}
sub foo {
    my ($a)=67;
    local($b)=@_;
    bar($a, $Ь);
}
foo(5, 10)
```

1. Какое значение будет иметь переменная $Ь после выполнения оператора bar($a, $b)?

   а) 5;

   б) 100;

   в) 68.

2. Какое значение возвращается функцией foo()?

   а) 67;

   б) 68;

   в) undef.

3. Какую область видимости имеет переменная $Ь в функции foo{)?

   а) лексическую;

   б) динамическую;

   в) глобальную.

Ответы

1. Правильный ответ — вариант б). $b — локальная переменная функции foo(), поэтому каждая вызываемая из нее подпрограмма может использовать эту переменную (если в ней не объявляется новая переменная с таким же именем с помощью операторов local или my). После вызова функции bar() значение переменной $b изменяется и становится равным 100.

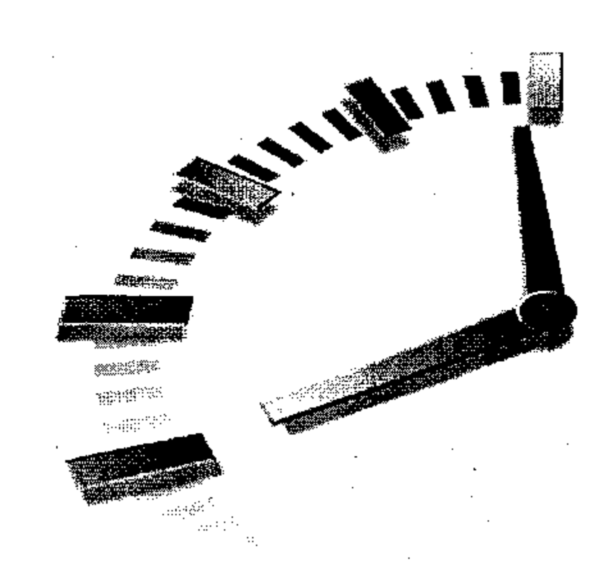
2. Правильным является вариант б). Удивлены? Последнее выражение функции foo() — bar($a,$b) возвращает 68, так как значение $a передается в bar() и там инкрементируется. Функция foo() возвращает значение своего последнего выражения — 68.

3. Правильный ответ — вариант б). Переменные, объявленные с помощью оператора local, называются переменными с динамической областью видимости.

Упражнения

- Используйте функции упражнения по статистике, приведенные выше, и код подсчета слов, рассмотренный на 7-м занятии, "Хэши", для анализа длины слов в документе. Вычислите их среднюю длину, медиану и стандартное отклонение длины.

- Напишите функцию для вывода части последовательности чисел Фибоначчи. Числа Фибоначчи — это бесконечная математическая последовательность, подсказанная самой природой. Она начинается с 0, 1, I, 2, 3, 5, 8. Каждый последующий ее член является суммой двух предыщуших (кроме 0 и 1). Эти числа могут быть вычислены с помощью рекурсивной процедуры или цикла.



Часть II

Углубляемся в Perl

Темы занятий

9-й час

Дополнительные функции и операторы

В культуре Perl есть такая традиция: "Существует более одного способа сделать что-либо". На этом занятии мы познакомимся с данной философией поближе. Мы рассмотрим целую "сборную солянку" новых функций и операторов.

При поиске скалярных величин и выполнении операций над ними вы до сих пор использовали регулярные выражения. Но поскольку в Perl существует более одного способа сделать что-либо, в нем предусмотрены различные функции поиска и редактирования скаляров. Итак, на этом занятии вы познакомитесь с некоторыми новыми способами.

Кроме того, раньше мы рассматривали массивы как линейные списки элементов, перебор которых осуществлялся с помощью оператора foreach, а объединение в скалярную переменную — с помощью функции join. На этом занятии вы познакомитесь с совершенно новым взглядом на массивы.

И, наконец, мы вновь обратимся к абсолютно "пресной" функции print и добавим ей немного остроты и привлекательности. С помощью обновленной и улучшенной функции print вы тоже сможете создавать тщательно отформатированные отчеты, которые не стыдно показать другим людям.

Основные темы этого занятия.

- Поиск скаляров внутри простых строк.
- Подстановка символов.
- Использование функции print.
- Применение массивов в качестве стеков и очередей.

Поиск скаляров

Регулярные выражения — прекрасный способ поиска скаляров по шаблонам, но иногда это просто стрельба из пушки по воробьям. В **Perl** некоторые "накладные расходы" (впрочем, небольшие) связаны с трансляцией шаблона, а затем поиском этого

шаблона среди скаляров. Кроме того, при написании регулярных выражений легко ошибиться. В Perl предусмотрено несколько функций поиска и извлечения простой информации из скаляров.

Функция index

Если вы просто хотите найти одну строку внутри другого скаляра, воспользуйтесь предусмотренной для этого в Perl функцией index. Ее синтаксис выглядит следующим образом:

index *строка, подстрока*
index *строка, подстрока,* **начальная_позиция**

Функция index начинает просмотр *строки* слева и ищет в ней *подстроку.* Функция index возвращает позицию, в которой найдена *подстрока,* причем значение 0 соответствует крайней левой позиции в строке. Если *подстрока* не найдена, то функция index возвращает значение -1. Искомая строка может состоять из букв, быть скаляром или любым выражением, возвращающим строковое значение. *Подстрока* — это не регулярное выражение, а просто другой скаляр.

Ниже приведено несколько примеров. Помните, что функции и операторы Perl можно записывать, заключая аргументы в круглые скобки или не делая этого.

```
index "В лесу родилась елочка", "елочка"; # Возвращается 16
index("Жить - хорошо, а хорошо жить - еще лучше", "еще"); # Возвращается 29
$a="Новое - это хорошо забытое старое";
index($a, "это");  I Возвращается 8
index($a, "новое"); I Возвращается -1
@a=qw(пшеница рис ячмень);
index join(" ", Ha), "рис"; # Возвращается 8
```

Можно также (хотя это и необязательно) указать при вызове функции index начальную позицию в строке, с которой нужно начать поиск, как показано в следующем примере. Чтобы выполнить поиск с начала строки (с крайней левой позиции), используйте начальную позицию 0.

```
$names="Герц Герцен Герцеговина Герцигова";
index($names, "Герц") ;# Возвращается 0
index($names, "Герц",1);# Возвращается 5
```

Можно также использовать функцию index с начальной позицией, чтобы "пройти" всю строку и найти все случаи вхождения в нее подстроки, как показано в следующем примере:

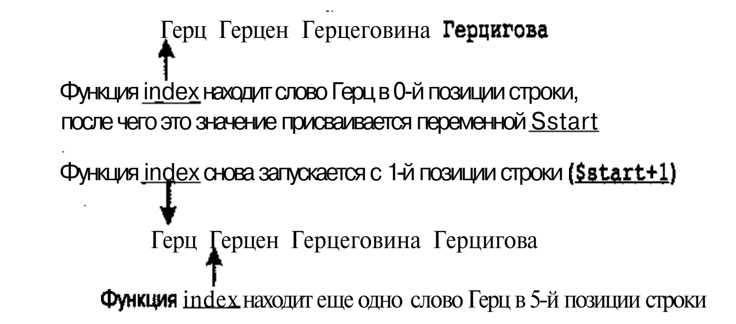
```
$source="Герц Герцен Герцеговина Герцигова";
$start=0;
f Чтобы найти все вхождения слова "Герц", мы будем
# использовать переменную $start в качестве начальной
# позиции для поиска. При этом значение переменной $start
# будет все время увеличиваться
while(($start=index($source,"Герц",$start)) !=-1){
   print "Найдено слово Герц в $start-й позиции\n";
   $start++;
}
```

В результате выполнения данного примера будет выведено следующее:

```
Найдено слово Герц в 0-й позиции
Найдено слово Герц в 5-й позиции
```

Найдено слово **Герц** в 12-й позиции
Найдено слово Герц в 24-й **позиции**

Функция index из предыдущего примера выполняет в цикле сканирование строки **$source**, как показано ниже:



Поиск в обратном направлении с помощью функции rindex

Функция rindex работает точно так же, как index, за исключением того, что поиск начинается с крайнего правого символа строки и проводится в левую сторону. Синтаксис данной функции выглядит следующим образом.

rindex *строка, подстрока*
rindex *строка, подстрока, начальная_позиция*

Если поиск закончен, а *подстрока* не найдена, то функция rindex возвращает значение -1. Ниже приведено несколько примеров.

```
$a= "Мишка косолапый по лесу идет, шишки собирает...";
rindex($a, "ишк"); # Возвращает 31
rindex($a, "ишк", 30); I Возвращает 1
```

По сравнению с функцией index, код для реализации обратного поиска в цикле с помощью функции rindex выглядит несколько иначе. Причина заключается в том, что функция rindex начинает поиск с правого крайнего символа строки. Поэтому в качестве начальной позиции для поиска следует указать символ, расположенный *за* последним символом строки. Для этого вполне подойдет значение, возвращаемое функцией length($source), как показано в следующем примере. Выполнение цикла следует завершить как только функция rindex возвратит значение -1. После нахождения каждого очередного элемента значение переменной $start должно быть уменьшено на 1, а не увеличено, как в случае использования функции index.

```
$source="Герц Герцен Герцеговина Герцигова";
$start=length($source);
while(($start=rindex($source,"Герц",$start)) !=-1){
   print "Найдено слово Герц в $start-й позиции\n";
   $start--;}
```

Выделение части строки с помощью функции substr

О функции substr часто забывают, не придавая ей особого значения, хотя она предоставляет универсальный метод выборки информации из скаляров и их редактирования. Синтаксис этой функции выглядит следующим образом:

substr *строка, смещение*
substr *строка, смещение,* **длина**

Функция substr берет *строку,* начиная с позиции, указанной во втором параметре, и возвращает оставшуюся часть строки (т.е. начиная с позиции *смещение* и до конца). Если параметр *длина* задан, то берется подстрока соответствующей длины. Если же в результате происходит выход за пределы строки, то просто берется подстрока до конца строки. Это продемонстрировано в следующем примере.

```
#Позиция символов в строке $a
#  0         10        20        30        40        50
$a="0 сколько нам открытий чудных готовит просвещенья век";
print substr($a, 50);# Возвращается "век"
print substr($a, 10, 12);# Возвращается "нам открытий"
```

ЕСЛИ задано отрицательное значение параметра *смещение,* то функция substr начинает отсчет справа. Например, substr($a, -5) возвращает пять последних символов строки $a. Если задано отрицательное значение параметра *длина,* то функция substr возвращает подстроку от начальной позиции и до конца строки, за исключением последних символов, количество которых определяется параметром *длина.* Например:

```
print substr($a, 30, -4);# Возвращается "готовит просвещенья"
```

В предыдущем фрагменте функция substr начала работу с 30-й позиции и вернула оставшуюся часть строки за исключением последних 4-х символов.

Функцию substr можно использовать также в левой части оператора присваивания. Такая конструкция позволяет указать, какие символы в скаляре должны быть заменены. В качестве первого аргумента функции substr следует задать скалярную переменную, а не строку символов, которой будет присваиваться значение. Ниже приведен пример редактирования строки с помощью функции substr,

```
$a="Все выше и выше, и выше";
# Заменим первое слово "выше" на "ниже"
substr($a, 4, 4)="ниже";

# Вставим в начало строки слово "Песня:"
substr($a, 0, 0)="Песня:";

# Заменим последние 4 символа
substr($a, -4, 4)="ниже";
```

Транслитерация, а не подстановка

Прежде чем рассмотреть оператор *транслитерации* (который иногда называют оператором ***перекодировки***), давайте немного вспомним, как работает оператор, подстановки, о котором шла речь на 6-м занятии, "Поиск по шаблону". Оператор подстановки

имеет вид **s/шаблон/замена/** и, если не задан оператор привязки =", выполняет поиск по шаблону и замену строки, находящейся в служебной переменной $_. Оператор транслитерации чем-то напоминает оператор подстановки, однако работает он немного иначе, поскольку в нем не используются регулярные выражения. Синтаксис этого операторы выглядит так:

tr/список_поиска/список_*замены/*

Оператор транслитерации tr/// выполняет поиск в строке элементов, указанных в первом списке, и заменяет их на соответствующие элементы из второго списка. По умолчанию поиск и замена выполняются в строке, находящейся в переменной $_. Чтобы изменить это правило, необходимо воспользоваться оператором привязки =~, как в случае использования регулярных выражений, например:

```
tr/ABC/XYZ/; # В переменной $_ все буквы "A" заменяются на "X",
             # "B" на "Y" и т.д.
$r=~tr/ABC/XYZ/; # Та же операция выполняется над переменной $r
```

Для логической группировки символов используется дефис. Например, конструкция A-Z представляет список прописных букв от A до Z включительно. Логическая группировка позволяет избежать перечисления символов в списке, например:

```
tr/A-Z/a-z/;       # Изменяет регистр букв с верхнего на нижний
tr/A-Za-z/a-zA-Z/; # Изменят регистр букв на обратный
```

Если в операторе транслитерации второй список идентичен первому или вовсе отсутствует, то оператор tr/// выполняет только подсчет найденных символов и возвращает данное значение. При этом исходная строка не изменяется, например:

```
$eyes=$potato=~tr/i//; # Выполняется подсчет количества символов "i"
                       #  в переменной $potato и найденное значение
                       #  присваивается переменной $eyes
$nums=tr/0-9//;        # Выполняется подсчет количества цифр
                       #  в переменной $_ и найденное значение
                       #  присваивается переменной $nums
```

И в заключение отметим, что у оператора tr/// есть синоним — оператор y///. Исторически так сложилось, что префиксы tr и y обозначают одну и ту же операцию. В операторе tr/// (а значит, и в y///) можно заменить символ разделитель списков. Обычно программисты выбирают пару круглых или квадратных скобок или любой другой подходящий символ, например:

```
tr(a-z)(n-za-m);  # Выполнение циклического сдвига влево на 13
                  # символов содержимого переменной $_.
y[,._-][;:=|]     # Замена некоторых символов пунктуации
```



> У оператора tr/// есть ряд дополнительных функциональных возможностей, которые используются сравнительно редко. Чтобы ознакомиться с ними, обратитесь к разделу prelop справочного руководства.

Улучшение качества печати

Выходные данные, которые мы до сих пор выводили на печать с помощью функции print, имели незатейливый вид. Дело в том, что функция print предназначена для выполнения отладочной печати, поэтому в ней не предусмотрено практически никаких средств

форматирования. Для получения качественных распечаток следует воспользоваться другой функцией Perl — **printf**. В ней предусмотрен широкий набор средств управления внешним видом выводимых данных, таких как выравнивание по левому и правому краю поля, изменение количества знаков после десятичной точки, получение полей фиксированной ширины и др. Функция printf была почти полностью заимствована из языка программирования С, однако стоит отметить, что в других языках программирования имеется аналогичная по своим функциональным возможностям функция (например, print using в BASIC). Синтаксис функции printf выглядит следующим образом:

printf***шаблон_форматирования,*** *список*
printf***дескриптор_файла шаблон_форматирования,*** *список*

Параметр ***шаблон_форматирования*** предназначен для описания формата выводимых данных в сокращенном виде. Вместо *списка* следует указать значения, выводимые функцией printf на печать (так же, как и в операторе print). По умолчанию вывод осуществляется в стандартный выходной поток (дескриптор **STDOUT**), однако, как и в случае функции print, можно указать дескриптор файла, в который следует поместить данные. Обратите внимание, что между дескриптором файла и шаблоном форматирования *не должно* быть запятой.

Шаблон форматирования обычно задается в виде литерала (реже в виде скалярной переменной) и определяет внешний вид выводимых данных. Любые символы, указанные в шаблоне, кроме тех, что начинаются с **%**, помещаются в неизменном виде в выходной поток. Символ процента обозначает начало *спецификатора поля,* который задается в виде ***%-w.dx*** (рис. 9.1). Параметр ***w*** **задает** ширину поля в символах; параметр *d* определяет количество цифр после десятичной точки (для числовых данных) или общую допустимую ширину поля для строк; параметр *x* обозначает тип выводимых данных. Дефис перед параметром ***w*** означает, что данные в поле шириной *w* символов выравниваются по его левому краю. По умолчанию данные будут выровнены по правому краю поля. Обязательными в спецификаторе поля являются только символ процента и поле *x*. Список некоторых часто используемых спецификаторов типа поля указан в табл. 9.1.

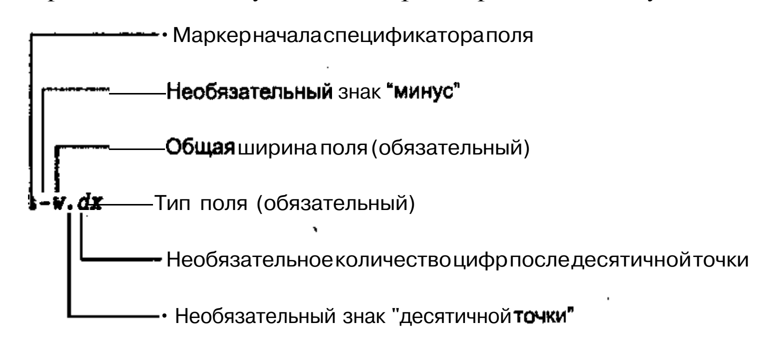


Рис. 9.1. Формат спецификатора поля

Таблица 9.1. Некоторые спецификаторы типа поля функции

Спецификатор типа	Тип
c	Символ
s	Строка
d	Десятичное целое; дробная часть числа отбрасывается
f	Число с плавающей точкой

Полный список спецификаторов типа поля можно найти в электронном справочном руководстве по Perl. Для этого введите команду perldoc -f **printf**.

Ниже приведено несколько примеров использования функции printf.

```
printf("%20s", "Джек");  # Выводит слово "Джек", которое
                         # располагается в поле шириной 20 символов в.
                         # выровнено по правому краю
printf("%-20s", "Джек"); # То же, только выравнивание выполняется по
                         # левому краю поля
Samt=7.12;
printf("%6.2f", $amt);   # Выводит "  7.12"
$amt=7.127;
printf("%6.2f", $amt);   # Выводит "  7.13", число округляется
printf("%c", 65);        # Выводит ASCII-символ,
                         # соответствующий коду 65 ("A")
$amt=9.4;
printf("%6.2f", $amt);   # Выводит "  9.40"
printf("%6d", $amt);     # Выводит "     9"
```

При выполнении форматирования каждый спецификатор поля заменяется соответствующим элементом из списка, как показано на рис. 9.2. При этом для каждого элемента в списке должен быть предусмотрен свой спецификатор поля и, наоборот, каждому спецификатору поля должен соответствовать элемент в списке.

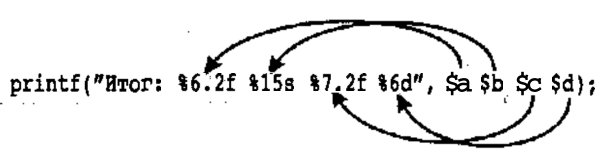


Рис. 9.2. Порядок замены спецификаторов поля

Чтобы вывести перед числом незначащие нули, нужно поместить символ 0 в спецификатор формата, как показано ниже:

```
printf("%06.2f", $amt); * Выводит "009.40"
```

Кроме функции printf, существует близкая по смыслу функция sprintf. Разница между ними состоит в том, что функция sprintf не выводит данные в выходной поток, а вместо этого просто их возвращает в вызвавшую ее программу. В результате форматированную строку данных можно присвоить скалярной переменной или использовать в любом другом выражении, как показано ниже на примере:

```
$weight=85;
t Округлим результат до 2-х цифр после запятой
$moonweight=sprintf("%.2f", $weight*.17);
print "Масса тела на Луне составит $moonveight кг.";
```

При выполнении форматирования не забывайте, что функции printf и sprintf, в параметрах которых указан спецификатор поля **%f**, автоматически округляют результат так, чтобы он поместился в отведенное ему поле.

Упражнение: создание отчета

В этом разделе мы рассмотрим одну из задач, с которой неизбежно сталкивается любой человек, работающий с компьютером. Речь идет об обработке исходных данных, их форматировании и составлении отчета. Эта проблема возникла не на пустом месте. Все

дело в том, что компьютеры и люди оперируют данными, представленными в разных форматах. В результате без специальной обработки человеку очень трудно проанализировать машинные данные. Поэтому наша задача — отформатировать данные, находящиеся в компьютере, так, чтобы с ними было удобно работать человеку.

Для выполнения этого упражнения нам понадобится набор записей о гипотетических сотрудниках некой компании, в который входит: фамилия и имя сотрудника, его табельный номер, количество отработанных им часов и почасовая тарифная ставка. Программа должна на основе этих данных составить отчет по зарплате.

Вы легко можете модифицировать исходный текст данной программы так, чтобы в результате получить любой необходимый вам отчет. Для упрощения упражнения записи с исходными данными мы будем хранить в массиве, который инициализируется в начале программы. Очевидно, что при построении реального отчета программа должна будет прочитать данные, хранящиеся в одном из файлов на диске. Однако усовершенствованием нашей программы мы займемся чуть позже.

Итак, запустите свой любимый текстовый редактор, наберите в нем программу, приведенную в листинге 9.1, и сохраните ее на диске в файле с именем Employee. Номера строк вводить не нужно. После сохранения программы сделайте файл выполняемым, как было описано на 1-м занятии, "Начало работы с Perl".

После выполнения подготовительных операций запустите программу, набрав в командной строке

```
perl Employee
```

Результаты работы программы приведены в листинге 9.2.

Листинг 9.1. Исходный текст программы Employee

```
#!/usr/bin/perl

use strict;

my @employees=(
     'Смит,Боб,123101,9.35,40',
     'Франклин,Алиса,132912,10.15,35',
     'Войоховиц,Тед,198131,6.50,39',
     'Нег,Венди,141512,9.50,40',
     'Клиборн,Стен,131211,11.25,40',
);

sub print_emp {
     my ($last,$first,$emp,$hourly,$time)=
               split(',',$_[0]);
     my $fullname;
     $fullname=sprintf("%s %s", $first, $last);
      printf("%6d %-20s %6.2f %3d %7.2f\n",
          $emp, $fullname, $hourly, $time,
           ($hourly * $time>+.005);
}

@employees=sort {
          my ($L1, $F1)=split(',', $a);
          my ($L2, $F2)=split(',', $b);
          return ( $L1 cmp $L2 \ Сравним фамилии
                         ||      # Если они идентичны...
```

```
28:                  $F1 cmp $F2 # Тогда сравним имена
29:                };
30:              }@employees;
31:
32: foreach(@employees) {
33:     print_emp($_);
34: >
```

Листинг 9.2. Результат работы программы Employee

```
198131 Тед Войоховиц        6.50  39  253.50
131211 Стен Клиборн        11.25  40  450.00
141512 Венди Нег            9.50  40  380.00
123101 Боб Смит             9.35  40  374.00
132912 Алиса Франклин      10.15  35  355.25
```

; Проведем анализ программы.

- *Строка 7.* В этой строке указывается путь к интерпретатору (измените его в соответствии с конфигурацией вашей системы) и ключ `-w`. Всегда включайте режим вывода предупреждений!
- *Строка 3.* Директива use `strict` указывает, что все переменные должны быть явно объявлены в программе с помощью оператора `my` и строки должны быть заключены в кавычки.
- *Строки 5-11.* Список сотрудников вместе с исходными данными присваивается массиву `@employees`. Каждый элемент массива имеет формат: фамилия и имя сотрудника, его табельный номер, величина часовой тарифной ставки и количество отработанных часов в неделю.
- *Строки 23~30.* Элементы массива `@employees` сортируются сначала по фамилии сотрудника, а затем по имени.
- *Строка 24.* Первый сортируемый элемент ($a) разбивается на части по полям. При этом фамилия присваивается переменной `$L1`, а имя— $F1. Обе эти переменные описаны в блоке как приватные с помощью оператора `my`.
- *Строка 25.* Те же действия выполняются над вторым сортируемым элементом (`$b`). При этом фамилия присваивается переменной $L2, а имя — $F2.
- *Строки 26~29.* Имена сравниваются в алфавитном порядке. Синтаксис оператора sort был рассмотрен на 4-м занятии, "Укладка строительных блоков: списки и массивы".
- *Строки 32—34.* В цикле выполняется перебор элементов отсортированного массива `@employees`. Каждый элемент передается функции `print_emp()`.
- *Строки 13—21.* Функция print_emp() выполняет форматирование и печать одной строки отчета.

- *Строки 14–15.* Переданная функции `print_emp()` строка находится в переменной `$_[0]`. Она разбивается на отдельные поля, и полученные значения для удобства обращения присваиваются переменным $last, `$first` и т.д., которые являются приватными для функции `print_emp()`.
- *Строка 17.* Фамилия и имя сотрудника объединяются в одну строку с помощью функции `sprintf`. В результате полученную строку можно будет легко поместить в поле фиксированной ширины и выровнять по левому краю.
- *Строки 18-20.* Строка отчета форматируется и выводится на печать. Для получения величины зарплаты количество отработанных часов (переменная `$time`) нужно умножить на почасовую тарифную ставку (переменная `$hourly`). Для выполнения правильного округления к двум значащим цифрам к произведению прибавляется значение .005.

Списки и стеки

До сих пор мы рассматривали списки и массивы как линейные структуры данных, доступ к элементам которых осуществлялся с помощью индексов (рис. 9.3).

0	1	2	3	4	5
Яблоко	Персик	Груша	Слива	Манго	Гуава

Рис. 9.3. Пример линейной структуры данных

А теперь включите свое воображение и представьте себе этот массив элементов в виде вертикальной стопки (рис. 9.4).

В компьютерной терминологии подобная структура данных называется *стеком.* Стеки обычно используются там, где необходимо выполнить обработку элементов списка в определенном порядке. Хорошим примером использования стека могут служить компьютерные карточные игры. В них каждый расклад карт представляется в виде стека. Сначала карты в колоде расположены рубашкой вверх. По мере необходимости карты переворачиваются и извлекаются из колоды (т.е. из стека). Кроме того, при отбое отыгравшие карты складываются в верхнюю часть колоды.

В Perl элементы стека обычно хранятся в массиве. Для помещения элемента в вершину стека используется функция push, а для извлечения элемента из стека — функция pop. Кроме того, можно поместить элемент в нижнюю часть стека и удалить его оттуда — по аналогии с колодой карт. Для этого используются функции shift и unshift соответственно. Операции со стеком проиллюстрированы на рис. 9.5.

Синтаксис перечисленных выше функций выглядит так:

pop *массив*
shift *массив*
unshift *массив, новый_список*
push *массив, новый_список*

Функции pop и shift удаляют один элемент из *массива.* Если параметр *массив* не указан, то будет удален один элемент либо из массива `@_`, либо из `@ARGV`. Функции pop и shift возвращают в вызвавшую программу удаленный из массива элемент, а если массив пуст, то возвращается значение undef. При этом соответствующим образом уменьшается и размер массива.

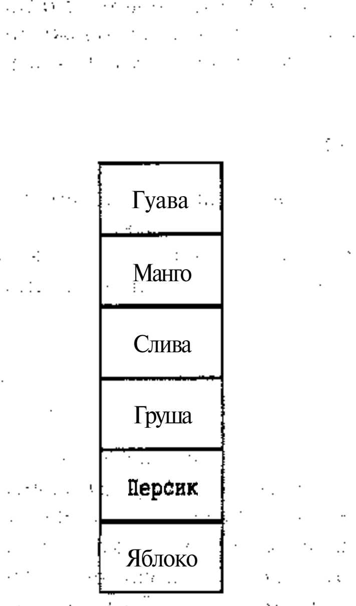


Рис. 9.4. Пример стека

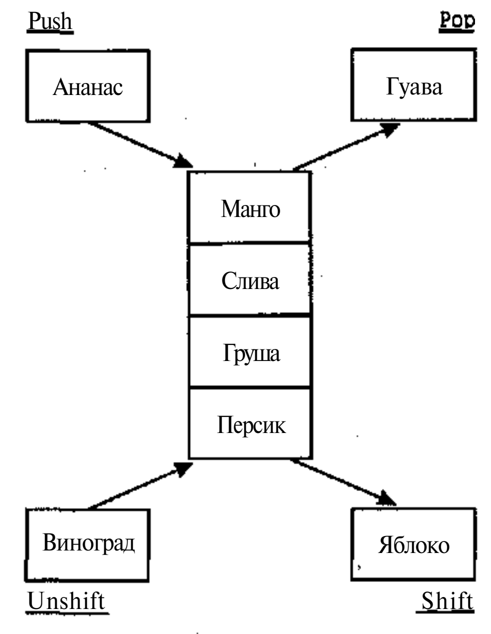


Рис. 9.5. Операции со стеком



> Внутри подпрограмм функции pop, shift, unshift и push модифицируют переменную @_, если не указан другой массив. В теле главной программы эти функции модифицируют массив @ARGV, если не указан другой массив.

Функции push и unshift добавляют элементы *нового_списка* в *массив*. При этом соответствующим образом увеличивается размер *массива*. Элементы *нового_списка* могут быть представлены в виде списка, массива или скаляра, как показано в следующем примере.

```
@band=qw(тромбон);
push @band, qw(гитара кларнет);
# В массиве @band сейчас содержится три элемента:
# "тромбон", "гитара", "кларнет"

$brass=shift 8band; I Переменная $brass равна "тромбон"
$wind=pop @band;    # Переменная $wind равна "кларнет"

\ В массиве @band сейчас содержится один элемент - 'гитара'
unshift @band, "аккордеон";
\ В массиве @band сейчас содержится два элемента:
t "аккордеон" и "гитара"
```



> Операции добавления и удаления элементов массива с помощью функций push, pop, shift и unshift гораздо эффективнее аналогичных операций, выполненных вручную. Например, код push @list,@newitems эффективнее, чем @list=(@list,@newitems). Перечисленные выше функции Perl специально оптимизированы для выполнения подобных операций над массивами.

При работе со стеком не стоит забывать и о том, что его элементы ничем не отличаются от элементов обычных массивов, которые можно адресовать с помощью индексов. При этом началу стека (или его нижней части) соответствует индекс 0, а вершине стека — последний элемент массива.

Слияние и разделение массивов

До сих пор мы ВЫПОЛНЯЛИ с массивами такие операции, как обращение к элементу по индексу, разделение массива на отдельные скаляры, а также всевозможные стековые манипуляции. Теперь пришла пора рассмотреть еще одну функцию — splice, с помощью которой можно выполнять как слияние, так и разделение массивов. Ее синтаксис выглядит так:

```
splice массив, смещение
splice массив, снещевве, длвва
splice массив, смещение,   длина, список
```

Функция splice удаляет из *массива* элементы, начиная с заданного *смещения*. При этом удаленные элементы возвращаются в вызывающую программу. Отрицательные значения *смещения* означают, что соответствующий элемент отсчитывается не с начала, а с конца массива. Если указан параметр *длина,* то удаляется указанное количество элементов массива. Если указан параметр *список,* то после удаления указанного количества элементов вместо них в *массив* помещаются элементы из *списка.* Во время выполнения подобных операций размер массива соответствующим образом уменьшается или увеличивается. Вот несколько примеров:

```
@veg=qw(морковь кукуруза);
splice (@veg, 0, 1);               # В массиве @veg находится один
                                   # элемент: "кукуруза"
splice {8veg, 0, 0, qw(горох));    # В массиве @veg находятся два
                                   # элемента: "горох", "кукуруза"
splice (@veg, -1, 1, qw(ячмень, репа)); # В массиве @veg находятся три
                                   # элемента: "горох", "ячмень", "репа"
splice (@veg, 1, 1);               # В массиве @veg находятся два
                                   # элемента: "горох", "репа"
```

Резюме

На этом занятии вы узнали, что для выполнения поиска одной строки в другой необязательно использовать регулярные выражения; для этого вполне достаточно функций index и rindex. Кроме того, вы научились выполнять простые подстановки с помощью оператора tr///. Для выделения части строки, а также для редактирования строки можно использовать функцию substr. Для создания отчетов, форматирования текстовых строк и округления чисел предусмотрены функции printf и sprintf. И в заключение были рассмотрены нелинейная структура данных (стек) и способы работы со стеком.

Вопросы и ответы

Можно ли обойтись без функций substr, index и rindex? Для чего они вообще нужны, если практически все можно сделать с помощью регулярных выражений?

Во-первых, использование регулярных выражений для выполнения простых видов поиска и замены строк крайне неэффективно. Функции index и rindex работают зна-

чительно быстрее. Во-вторых, создание регулярных выражений для выполнения замены текста с фиксированной позиции в строке — это стрельба из пушки по воробьям; намного более элегантное решение — воспользоваться функцией substr. И, наконец, в-третьих, Perl относится к тому разряду языков программирования, где одну и ту же операцию можно выполнить разными средствами. Поэтому используйте то средство, которое вам больше нравится.

Что произойдет, если в качестве начальной позиции в функции substr (или index, или rindex) указать значение, выходящее за границы строки?

Чем мне всегда нравились компьютеры, так это их удивительной терпимостью к разного рода экспериментам. Вместо того чтобы задавать вопрос из серии "Что будет, если...?", иногда проще самому поэкспериментировать и посмотреть, что получится.

Возвращаясь к поставленному вопросу, стоит отметить, что обращение к несуществующей части строки приведет к появлению сообщения об ошибке use of undefined value (если включен режим выдачи предупреждений, а он у вас должен быть включен всегда!). При этом функция возвращает неопределенное значение. Например, при выполнении кода $a="Foo"; `substr($a,5);` функция substr возвращает значение undef.

Семинар

Контрольныевопросы

1. Какое значение будет иметь переменная `@A` после выполнения следующего фрагмента кода?

```
@A=qw(овес горох бобы);
shift @A;
push @A, "ячмень";
pop;
```

   а) овес горох бобы;

   б) бобы ячмень;

   в) горох бобы ячмень.

2. Что делает оператор `printf("%18.3f",$a)`?

   а) выводит число с плавающей точкой, занимающее 18 позиций (15 позиций до десятичной точки и 3 после нее);

   б) выводит число с плавающей точкой, занимающее 18 позиций до десятичной точки и 3 после нее;

   в) выводит число с плавающей точкой, занимающее 18 позиций (14 позиций до десятичной точки и 3 после нее).

3. Предположим, что к некоторой строке был применен оператор tr/a-z/A-Z/. Восстановит ли первоначальное состояние строки оператор tr/A-Z/a-z/?

   а) да;

   б) скорее всего, нет.

Ответы

1. Правильный ответ — вариант в). Функция shift удаляет из массива элемент "овес", а следующая за ней функция push добавляет в конец массива элемент "ячмень". Последняя функция фрагмента pop является уловкой. Поскольку в ее параметрах не указан массив, она модифицирует переменную `@_`, которая никакого отношения не имеет к массиву `@A`.

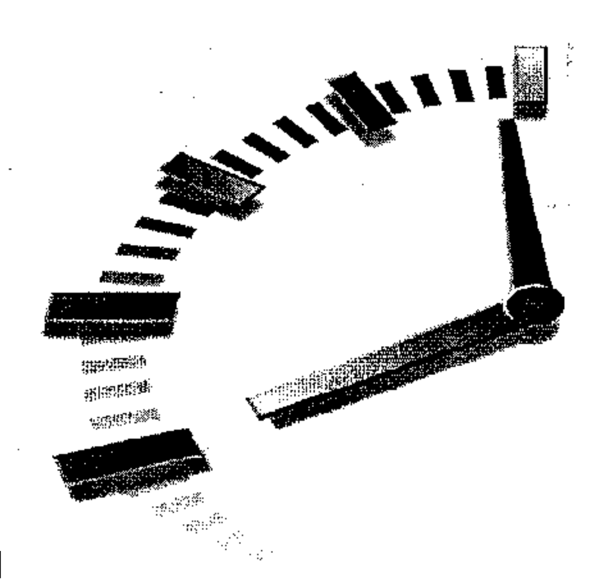
2. Правильный ответ — вариант в). Если вы выбрали вариант а), то не учли тот факт, что десятичная точка занимает одну позицию в строке размером 18 символов (18=14+1+3).

3. Правильным является ответ б). Например, строка "Rosebud" в результате перекодировки с помощью оператора `tr/a-z/A-Z/` будет иметь вид "ROSEBUD". Применение оператора tr/A-Z/a-z/ изменит строку на "rosebud", а ведь это не то, что было вначале.

Упражнения

- Перепишите программу `Hangman`, которая рассматривалась на 4-м занятии, "Укладка строительных блоков: списки и массивы", так, чтобы в ней вместо массивов использовались скаляры. Для изменения отдельных символов в строке воспользуйтесь функцией substr.

- Измените программу, приведенную в листинге 9.1, так, чтобы исходные данные она читала из файла. Для этого вместо оператора инициализации массива необходимо поместить операторы, открывающие файл, читающие из него данные в массив и закрывающие файл. Оставшуюся часть программы менять не нужно. При тестировании программы не забудьте создать файл с исходными данными.



10-й час

Файлы и каталоги

ДЛЯ хранения данных в любой операционной системе используются *файлы.* Способ хранения данных, выбор системы именования файлов и выполнение поиска нужных файлов зависит от типа конкретной *файловой системы,* которая является составной частью любой операционной системы. Обычно операционная система позволяет хранить файлы в виде логических групп, которые называются *каталогами,* или *папками.* В каталогах могут находиться файлы или другие (вложенные) каталоги.

Систему вложенных каталогов часто представляют в виде древовидной структуры. При этом любой файл является составной частью отдельного каталога, а сам каталог входит в другой (родительский) каталог. Кроме поддержки организационной структуры файлов, операционная система также сохраняет некоторую дополнительную информацию о самом файле, такую как время последнего обращения, время модификации, имя владельца, текущий размер и т.д. Описанная выше модель файловой системы применяется в большинстве современных операционных систем.

В компьютерах Macintosh, несмотря на некоторые терминологические разногласия (каталог верхнего уровня называется *томом,* а вложенные каталоги — *папками),* также используется данная модель файловой системы.

В языке Perl предусмотрены средства для доступа к структуре файловой системы, изменения ее и получения подробной информации о самом файле. Прототипы функций Perl для открытия, чтения и записи файлов взяты из операционной системы UNIX, однако, они прекрасно работают в любой операционной системе. Другими словами, все функции для работы с файлами в Perl обладают свойством *переносимости.* А это означает, что программа манипуляции с файлами будет работать без изменений в любой операционной системе, для которой существует интерпретатор Perl (при условии, что эта операционная система поддерживает иерархическую файловую структуру).

Основные темы этого занятия.

- Получение листинга каталога.
- Создание и удаление файлов.
- Создание и удаление каталогов.
- Получение информации о файле.

Получение листинга каталога

Прежде чем выполнять операции с каталогом, необходимо создать его *дескриптор*. Дескриптор каталога напоминает дескриптор файла с тем лишь отличием, что при чтении считывается не файл, а содержимое каталога. Для открытия дескриптора каталога используется функция opendir:

opendir *имя_дескриптора, имя_каталога;*

В этом синтаксисе вместо *имени_дескриптора* следует указать дескриптор каталога, который вы хотите открыть, а вместо *имени_каталога* — путь к каталогу, содержимое которого нужно прочитать. Если по какой-либо причине каталог нельзя открыть (например, у вас отсутствуют права на доступ к нему, либо просто указано имя несуществующего каталога), функция opendir возвращает значение false. При выборе имени дескриптора каталога следует руководствоваться теми же правилами, что и при выборе имени дескриптора файла (о них шла речь на 2-м занятии, "Строительные блоки Perl: числа и строки"). Всегда набирайте имя дескриптора прописными буквами, чтобы исключить возможный конфликт имен с ключевыми словами Perl, которые появятся в будущих версиях этого языка программирования. Вот пример использования функции opendir:

```
opendir(TEMPDIR, '/tmp') {{ die "Ошибка при открытии каталога /tmp: $!";
```



> Во всех примерах этого занятия в качестве разделителя пути мы используем косую черту (/), как это принято в системе UNIX. Причина заключается в том, что такой стиль написания улучшает читабельность листингов (поскольку нет конфликтов со служебным символом Perl \) и не нарушает работоспособность программ в системе Windows.

После открытия каталога доступ к его содержимому можно получить с помощью функции readdir:

readdir *дескриптор;*

В скалярном контексте функция readdir возвращает следующий по порядку элемент каталога или значение undef, если достигнут конец каталога. В контексте списка функция readdir возвращает все оставшиеся элементы каталога. Имена, возвращаемые данной функцией, могут относиться как к файлам, так и к каталогам, а в системе UNIX — еще и к специальным файлам. Порядок их следования соответствует физическому расположению в каталоге. Другими словами, элементы каталога никак не сортируются. Кроме того, функция readdir возвращает еще два специальных элемента каталога: . и .., которые соответствуют текущему и родительскому каталогам. В элементы каталога не включается путь.

После завершения работы с каталогом его дескриптор следует закрыть с помощью функции closedir:

closedir *дескриптор',*

В следующем примере продемонстрирована методика чтения каталога:

```
opendir(TEMPDIR, '/tmp') ||
    die "Ошибка при открытии каталога /tmp: $!";
@FILES=readdir TEMPDIR;
closedir(TEMPDIR);
```

Здесь все содержимое каталога помещается в массив @FILES. Однако, чаще всего, из этого списка нужно исключить некоторые имена, например . и ..., поскольку для пользователя в них нет особого смысла. Для этого следует воспользоваться таким оператором чтения каталога:

```
@FILES=grep(!/^\.\.?$/, readdir TEMPDIR);
```

В этом примере регулярное выражение /^\.\.?$/ соответствует строке текста, в которой находится как минимум одна точка. Функция grep отфильтровывает такие строки, поскольку перед регулярным выражением стоит оператор отрицания. Если нужно отобрать элементы каталога, содержащие заданное расширение, оператор чтения каталога будет выглядеть так:

```
@FILES=grep(/\.txt$/i, readdir TEMPDIR);
```

Имена файлов, возвращаемые функцией readdir, не содержат путь, который мы указывали в функции opendir при открытии каталога. Поэтому следующий пример, скорее всего, не будет работать:

```
opendir(TEMPDIR, '/tmp') ||
     die "Ошибка при открытии каталога /tmp: $1";
while($file=readdir TEMPDIR) {
   # Приведенный ниже код неверен!!!
   open(FILEH, $file) ||
     die "Ошибка при открытии файла $file: $!\n";
    ...

}
closedir(TEMPDIR);
```

Кроме случаев, когда вы запускаете данную программу из каталога /tmp, при выполнении оператора open(FILEH, $file) будет возникать ошибка. Причина состоит в том, что программа читает список файлов каталога /tmp, а оператор open пытается открыть файл в текущем каталоге. Естественно, что если текущим является не каталог /tmp и имена файлов текущего каталога и каталога /tmp не совпадают, то функция open не будет находить файлы. Для решения проблемы в операторе open следует указать полный путь к файлу. Правильный код будет выглядеть так:

```
opendir(TEMPDIR, '/tmp') ||
     die "Ошибка при открытии каталога /tmp: $!";
while($file=readdir TEMPDIR) {
   # А этот код уже правильный!
   open(FILEH, "/tmp/$file") j|
     die "Ошибка при открытии файла $file: $!\n";
    ...

}
closedir(TEMPDIR);
```

Отбор файлов заданного типа

Существует еще один метод получения списка нужных файлов заданного каталога, который называется *отбором файлов (globbing)*. Если вы хоть немного работали с командной строкой DOS, то наверняка вам приходилось вводить команды наподобие dir *.txt. В данном случае команда dir выводит список всех файлов, имена которых

имеют расширение *.txt. В UNIX понятие расширения файла отсутствует, однако отбор нужных файлов также можно осуществить с помощью командной оболочки. Например, аналог приведенной выше команды dir в UNIX выглядит так: `ls *.txt`. В результате будет получен список всех файлов, имена которых оканчиваются суффиксом `.txt`.

В Perl также предусмотрен специальный оператор glob, выполняющий описанные действия. Его синтаксис выглядит так:

glob *шаблон*

Здесь параметр *шаблон* соответствует именам файлов, которые необходимо отобрать. Он может содержать путь, а также часть имени файла. Кроме того, в шаблоне может быть указано несколько специальных символов, описанных в табл. 10.1. В контексте списка оператор glob возвращает имена всех файлов и каталогов, имена которых соответствуют шаблону. В скалярном контексте при каждом вызове описываемый оператор возвращает имя следующего файла, соответствующего шаблону.



Не путайте шаблоны оператора glob с регулярными выражениями. Учтите, что это не одно и то же.

Таблица 10.1. Шаблоны оператора glob

Символ	Соответствует...	Пример
?	Одному символу	Шаблон f?d соответствует именам *fud, fid, fdd* и т.д.
*	Любому количеству символов	Шаблон f*d соответствует именам *fd, fdd, food, filled* и т.д.
[символы]	Любому символу, указанному в списке. Данная возможность не поддерживается в MacPerl	Шаблон f[ou]d соответствует именам *fod* или fud, но не *fad*
{строки}	Любой строке, указанной в списке. Данная возможность не поддерживается в MacPerl	Шаблон f*.{txt,doc} соответствует именам, которые начинаются на букву *f* и заканчиваются суффиксом `.txt` или `.doc`



Это замечание относится к приверженцам UNIX. В операторе Perl glob реализована методика отбора файлов, принятая в оболочке C. Она немного отличается от методики, используемой в оболочке Bourne (или Korn). Данное замечание справедливо для всех платформ UNIX, на которых установлен Perl, независимо от того, в какой оболочке работает конечный пользователь. И хотя методики отбора файлов во многом схожи, отличия все же есть, главным образом в способе интерпретации символов шаблона * и ?. Поэтому будьте внимательны.

Ниже приведено несколько примеров отбора файлов.

```
# Выберем все файлы .h из каталога /usr/include
my @hfiles=glob('/usr/include/*.h');

# А теперь отберем текстовые файлы, в именах которых
# встречается 1999 год
my @curfiles=glob('*1999*.{txt,doc}');
```

```
# Распечатаем нумерованный список файлов
$count=1;
while($name=glob('*')) {
   print "$count. $name\n";
   $count++;
}
```

Ниже приведен список основных отличий функции glob от opendir/readdir/closedir.

- Функция glob может возвращать только ограниченное количество файлов. Если в каталоге будет находиться большое количество файлов, эта функция, скорее всего, аварийно завершит свое выполнение. Причина состоит в том, что в текущей версии Perl функция glob реализована с помощью сценария оболочки С, который может возвращать только ограниченное количество файлов. При использовании функций opendir/readdir/closedir подобная проблема не возникает.

- Функция glob возвращает имя файла вместе с путем, который указан в шаблоне, тогда как функции `opendir/readdir/closedir` возвращают только имя файла. Например, оператор glob(`'/usr/include/*.h'`) к каждому возвращаемому имени файла добавляет путь `/usr/include/`.

- Функция glob работает медленнее, чем opendir/readdir/closedir. Причина очевидна. Perl должен запустить внешнюю программу, которая выполнит отбор и сортировку файлов, а затем получить от нее данные и интерпретировать их.

Итак, исходя из этого, какими же средствами лучше всего воспользоваться для отбора файлов? Ответ один — теми, которыми вам удобнее. Однако стоит иметь в виду, что использование opendir/readdir/closedir позволяет создать более универсальный и гибкий код. Поэтому в большинстве примеров мы используем именно набор функций opendir/readdir/closedir.

Для полноты картины стоит упомянуть еще об одном способе отбора файлов. Просто поместите шаблон в угловые скобки (о); в результате угловой оператор превратится в некое подобие оператора glob, например:

```
@cfiles=<*.c>; # Отобрать все файлы, имена которых
               # заканчиваются на .c
```

Синтаксис углового оператора для отбора файлов является устаревшим, к тому же он может ввести в заблуждение кого угодно, поэтому им пользоваться не рекомендуется. Для ясности в примерах из данной книги мы пользовались оператором glob.

Упражнение: реализация утилиты UNIX grep

Мы постарались так подобрать примеры, чтобы по мере чтения книги вы приобретали опыт в создании полезных программ. В этом упражнении мы реализуем усеченную версию популярной утилиты системы Unix grep. Эта утилита выполняет поиск файлов, содержимое которых соответствует заданному шаблону (не путайте ее с одноименным оператором Perl). В рассматриваемом нами примере имя каталога и шаблон для поиска файлов вводятся в диалоговом режиме по запросу программы. Затем программа выполняет поиск файлов по заданному шаблону и выводит их на печать.

Запустите текстовый редактор, наберите в нем программу, приведенную в листинге 10.1, и сохраните ее в файле **mygrep**. Не забудьте сделать файл выполняемым, как это было описано на 1-м занятии, "Начало работы с Perl". Убедитесь также, что вы случайно не присвоили файлу с программой на Perl имя grep (иначе в системе UNIX будет конфликт с утилитой grep).

Завершив подготовительные действия, запустите программу с помощью следующей команды:

```
perl mygrep
```

Листинг 10.1. Исходный текст программы mygrep

```
1:  #!/usr/bin/perl
2:
3:  use strict;
4:
5:  print "Введите имя каталога: ";
6:  my $dir=<STDIN>; chomp $dir;
7:  print "Введите шаблон для поиска: ";
8:  my $pat=<STDIN>; chomp $pat;
9:
10: my ($file);
11:
12: opendir(DH, $dir) || die "Ошибка при открытии каталога $dir: $!";
13: while($file=readdir DH) {
14:   next if (-d "$dir/$file");
15:   if (1 open(F, "$dir/$file")) {
16:         warn "Ошибка при открытии файла $file: $!";
17:         next
18:   }
19:   while(<F>){
20:         if (/$pat/J {
21:               print "$file: $_"
22:         }
23:   }
24:   close(F);
25: }
26: closedir(DH);
```

Проведем анализ программы.

- *Строка 1.* В этой строке указывается путь к интерпретатору (измените его в соответствии с конфигурацией вашей системы) и ключ **-w**. Всегда включайте режим вывода предупреждений!
- *Строка 3.* Директива use strict указывает, что все переменные должны быть явно объявлены в программе с помощью оператора **my** и строки должны быть заключены в кавычки.
- *Строки 5-8.* В диалоговом режиме вводятся значения переменных $dir (каталог для поиска) и Spat (шаблон для поиска). Символ перевода **строки** удаляется с помощью функции chomp.

- *Строка 10.* Поскольку в начале программы указан оператор use strict, то здесь объявляется переменная $file, которая будет использоваться ниже в программе.
- *Строка 12.* Открывается каталог для чтения, указанный в переменной $dir. Если во время выполнения этой операции происходит ошибка, выводится соответствующее сообшение.
- *Строка 13.* Элементы каталога считываются в цикле и помешаются в переменную $Ше.
- *Строка 14.* Элементы каталога, которые являются вложенными каталогами (оператор -d), пропускаются. Обратите внимание, что в имени файла указан путь ($dir/$file), поскольку текущий каталог может не совпадать с каталогом $dir. Таким образом, полный путь к файлу задается в виде "$dir/$file".
- *Строки 15—18.* Открывается файл, указанный с помощью полного пути. Если происходит ошибка, выводится соответствующее сообшение и обработка текущего файла прекращается.
- *Строки 19—23.* Файл сканируется построчно в поисках текста, соответствующего шаблону $pat. Найденные строки вместе с именем файла выводятся на печать.

Пример работы программы приведен в листинге 10.2.

Листинг 10.2. Пример работы программы mygrep

```
Введите имя каталога: .
Введите шаблон для поиска: perl
hello: #!/usr/bin/perl
lst02_01.pl: #!/usr/bin/perl -w
```

Каталоги

На этом занятии уже упоминалось о том, что каталоги на диске организованы в виде древовидной структуры, что для открытия файла иногда требуется указать его полный путь и что содержимое каталога можно прочитать с помощью функции readdir. Однако с помощью Perl можно еше перемещаться по каталогам, создавать и удалять их, а также удалять файлы, содержащиеся в каталогах (очищать каталоги).

Перемещение по каталогам

При запуске какой-либо программы операционная система "запоминает" текущий каталог, в котором находился пользователь перед вводом команды. В системе UNIX после регистрации пользователь обычно попадает в свой рабочий каталог. Чтобы узнать, в каком каталоге вы находитесь, в системе UNIX используется команда pwd. При работе в режиме командной строки MS-DOS или Windows путь к текущему каталогу отображается в виде приглашения, например C:\WINDOWS>. Кроме того, если в сеансе MS-DOS ввести команду cd без параметров, операционная система выведет на экран путь к текущему каталогу. Таким образом, каталог, используемый по умолчанию системой в настоящий момент, называется *текущим каталогом*.



Если вы работаете в одном из текстовых редакторов или интегрированной среде разработки и пытаетесь запустить на выполнение программу на Perl прямо из среды разработки, то текущий каталог, установленный операционной системой для вашей программы, может не соответствовать ожидаемому. Таким каталогом может являться каталог, в котором находится программа на Perl, рабочий каталог текстового редактора или любой другой каталог (в зависимости от установленных параметров среды разработки). Чтобы определить в программе на Perl рабочий каталог, воспользуйтесь функцией cwd.

Если при открытии файла не указан полный путь к нему, как, например, в операторе open(FH, "file") || die, будет предпринята попытка открыть указанный файл в *текущем* каталоге. Для изменения текущего каталога в программах на Perl используется функция chdir, как показано в следующем примере:

chdir ***новый_каталог***

Эта функция заменяет текущий каталог на тот, который указан в качестве параметра. Если указано имя несуществующего каталога или у вас отсутствуют права на доступ к новому каталогу, функция chdir возвращает значение false. С помощью этой функции можно *временно* изменить текущий каталог. Как только программа завершит свое выполнение, в качестве текущего будет восстановлен тот каталог, в котором находился пользователь перед запуском программы.

Вызов функции cbdir без параметров устанавливает в качестве текущего рабочий каталог пользователя. В системах UNIX это тот каталог, в который попадает пользователь после регистрации в системе. В Windows 9x, NT или MS-DOS рабочий каталог пользователя задается с помощью переменной окружения HOME. Если такая переменная не установлена, вызов функции chdir без параметров не выполняет никаких действий.

В Perl нет встроенных функций, которые бы позволяли определить текущий каталог программы. Причина заключается в том, что в силу специфики некоторых операционных систем сделать это достаточно сложно. Для решения задачи вам следует воспользоваться двумя связанными операторами. Укажите в начале программы оператор use Cwd, а затем для определения текущего каталога воспользуйтесь функцией cwd, как показано ниже на примере:

```
use Cwd;
print "Текущий каталог - ", cwd, "\n";
chdir '/tmp' or warn "Ошибка доступа к каталогу /tmp: $!";
print "Сейчас мы находимся в ", cwd, "\n";
```

Оператор use Cwd необходимо указывать в программе только один раз. После этого функцию cwd можно использовать столько раз, сколько нужно.



Оператор use Cwd указывает интерпретатору Perl, что в этом месте программы должен быть загружен модуль Cwd. *Модули* расширяют возможности Perl за счет добавления новых функций, таких как cwd. Если в предыдущем примере с использованием модулей интерпретатор выдаст ошибку типа Can't locate Cwd.pm in @INC или если вы так и не поняли, для чего вообще нужны модули, сильно не расстраивайтесь. Более подробно модули будут описаны на 14-м занятии, "Использование модулей".

Создание и удаление каталогов

ДЛЯ создания нового каталога в Perl используется функция mkdir, синтаксис которой выглядит так:

mkdir *новый_каталог, права_доступа',*

Если указанный каталог успешно создан, функция **mkdir** возвращает истинное значение. В противном случае она возвращает ложное значение, а текст сообщения об ошибке помещается в переменную $!. Параметр *права_доступа* имеет значение только в системах UNIX, однако его следует указывать всегда, независимо от того, работаете вы в UNIX или нет. В приведенных ниже примерах используется некое магическое число 0755, значение которого будет объяснено ниже, вразделе "Небольшой экскурс в UNIX". Для систем DOS и Windows просто используйте число 0755 и ни о чем не думайте! Сейчас мы не будем пускаться в долгие рассуждения, а сразу рассмотрим пример:

```
print "Укажите имя создаваемого каталога: •;
my $newdir=<STDIN>;
chomp $newdir;
mkdir($newdir, 0755) j| die "Ошибка при создании каталога $newdir: $!";
```

Для удаления каталога используется функция **rmdir**, синтаксис которой выглядит так:

rmdir *каталог;*

Если указанный каталог успешно удален, функция rmdir возвращает истинное значение. В противном случае она возвращает ложное значение, а текст сообщения об ошибке помещается в переменную $!, как показано в следующем примере:

```
print "Укажите имя удаляемого каталога: ";
my $baddir=<STDIN>;
chomp $baddir;
rmdir($baddir) || die "Ошибка при удалении каталога $baddir: $!";
```

С помощью функции rmdir можно удалить только *пустые* каталоги. Это означает, что перед удалением каталога в нем следует сначала удалить все файлы и вложенные каталоги.

Удаление файлов

ДЛЯ удаления файлов в **Perl** используется функция unlink, синтаксис которой выглядит так:

unlink ***список_файлов;***

Функция unlink пытается удалить все файлы, указанные в списке, и возвращает в вызывающую программу количество удаленных файлов. Если список файлов не указан, будет удален файл, указанный в переменной $_. Рассмотрим следующие примеры:

```
unlink <*.bat>;
$erased=unlink 'old.exe', 'a.out', 'personal.txt';
unlink @badfiles;
unlink; # Удаляется файл, имя которого указано в переменной $_
```

Чтобы проверить, действительно ли были удалены все файлы из списка, нужно сравнить количество файлов в списке с тем, которое вернет функция unlink, например:

```
my @files=<*.txt>;
my $erased=unlink §files;

f Сравним количество файлов в списке с тем,
# которое было удалено
if {$erased 1= @files) {
    print "Эти файлы не были удалены: ",
          join(', ', <*.txt>), "\n";
}
```

В этом примере число удаленных файлов сохраняется в переменной $erased. После выполнения функции unlink значение переменной $erased сравнивается с количеством элементов массива @files. Они должны быть одинаковыми. Если это не так, выводятся сообщение об ошибке и список тех файлов, которые не удалось удалить.



Файлы, удаленные с помощью функции unlink, восстановить уже нельзя, поскольку Perl не помещает их в корзину. Поэтому при работе с этой функцией будьте особенно аккуратны!

Переименование и перемещение файлов

Переименовать файл или каталог в Perl очень просто. Для этого используется функция rename, синтаксис которой приведен ниже:

```
rename старое_имя, новое_имя;
```

Эта функция берет файл, имя которого указано в первом параметре, и переименовывает его в файл, имя которого указано во втором параметре. Если операция завершается успешно, функция rename возвращает истинное значение. Если в качестве первого параметра указан каталог, ему будет назначено имя, заданное во втором параметре. Если работа функции rename завершается аварийно, она возвращает ложное значение, а текст сообщения об ошибке помещается в переменную $!, как показано в следующем примере:

```
if (! rename "myfile.txt", "arcfile.txt" ) {
    warn "Ошибка при переименования файла myfile.txt: $1";
}
```

С помощью функции rename можно также переместить файл в другой каталог. Для этого в качестве второго параметра укажите новый путь к файлу, например:

```
# Выполняется перемещение файла
rename "myfile.txt", "/tmp/myfile.txt";
```

Если окажется, что файл, заданный во втором параметре функции rename, существует, содержимое старого файла теряется.



С помощью функции rename нельзя переместить файлы из одного каталога в другой, если они принадлежат различным файловым системам.

Небольшой экскурс в UNIX

Данный раздел предназначен для тех, кто программирует на Perl в UNIX, если же вы не относитесь к этой категории людей, можете безболезненно пропустить его и перейти к следующему разделу. Однако приведенная здесь информация поможет вам разгадать тайну магических чисел, которые нужно указывать при создании каталога.

Начнем с того, что Perl имеет глубокие корни в UNIX. Более того, прототипом некоторых команд и операторов послужили аналогичные команды и вызовы операционной системы UNIX. С большинством таких функций вам никогда не придется сталкиваться. В то же время часть из них, как, например, функция unlink, используется достаточно часто. И несмотря на то что данная функция была также позаимствована из UNIX, большинство пользователей никак не связывает ее с миром UNIX. Причина заключает-

ся в том, что средства для удаления файлов есть в любой операционной системе, поэтому Perl делает так, чтобы функция unlink работала всегда, независимо от используемой ОС. Описанная концепция независимости от компьютерной платформы реализована в Perl практически повсеместно, в частности в функциях ввода/вывода. Интерпретатор по возможности делает так, чтобы все вопросы несовместимости были скрыты от пользователя. Все это делает программы на Perl действительно переносимыми.

Тот факт, что в Perl встроено большое количество команд и функций UNIX, которые перенесены на другие (не UNIX) компьютерные платформы, не может не радовать разработчиков и системных администраторов, привыкших работать в среде UNIX. Это позволяет им создать свой замкнутый мир UNIX, независимо от того, на компьютере какого типа они работают.

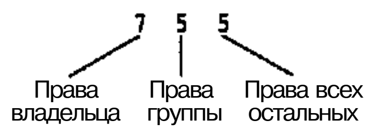


Как следует из названия следующего раздела, приведенное на страницах этой книги описание прав доступа к файлам системы UNIX и способов их назначения ни в коей мере нельзя считать полным. Чтобы получить исчерпывающую информацию по интересующим вас вопросам, обратитесь к документации по операционной системе или найдите одну из книг по UNIX, выпущенных *Издательским домом "Вильямс"*. За справками обращайтесь по адресу http://vwv.williamspublishing.com.

Немного о правах доступа к файлам

На 1-м занятии, "Начало работы с Perl", без всяких дальнейших объяснений говорилось, что, для того чтобы в UNIX запускать программу на Perl как обычную команду, к файлу, в котором находится эта программа, необходимо применить команду `chmod 755 имя_файла`. Магическое число 755 — это как раз и есть закодированные права доступа, которые назначаются указанному файлу. Таким образом, в системе UNIX для назначения и изменения прав доступа к файлам используется команда `chmod`.

Каждая из трех цифр представляет собой код прав доступа, которые назначаются владельцу этого файла, пользователям группы, к которой относится файл, и всем остальным пользователям компьютера (рис. 10.1). В нашем случае для владельца код прав доступа составляет 7, для группы — 5, а для всех остальных — 5. Коды прав доступа перечислены в табл. 10,2.



Рис, 10.1. Кодирование прав доступа к файлу

Таблица 10.2. Коды прав доступа к файлам

Код	Описание
7	Владелец/пользователи группы/остальные пользователи могут читать информацию из файла, изменять этот файл и запускать его на выполнение (RWX)
6	Владелец/пользователи группы/остальные пользователи могут читать информацию из файла и изменять файл (RW)
5	Владелец/пользователи группы/остальные пользователи могут читать информацию из файла и запускать его на выполнение (RX)
4	Владелец/пользователи группы/остальные пользователи могут читать информацию из файла (R)

Окончание табл. 10.2

Код	Описание
3	**Владелец/пользователи** группы/остальные пользователи могут записывать информацию в файл и запускать его на выполнение (**WX**)
2	**Владелец/пользователи** группы/остальные пользователи могут записывать информацию в файл (**W**)
1	**Владелец/пользователи** группы/остальные пользователи могут запускать файл на выполнение (**X**)

Для установки прав доступа к файлу в **Perl** используется встроенная функция **chmod**, синтаксис которой выглядит так:

```
chmod права_доступа, список_файлов;
```

Данная функция пытается изменить права доступа ко всем файлам, указанным в списке, и возвращает число файлов, для которых эта операция успешно выполнена. При указании кода *прав_доступа* не забудьте только поставить перед числом иифру О, поскольку права доступа задаются в виде восьмеричного литерала. Ниже приведено несколько примеров использования функции chmod.

```
chmod 0755, 'file.pl'      # Назначает права RWX для владельца и
                           # RX для группы в всех остальных
                           # пользователей
chmod 0644, 'mydata.txt'   # Назначает права RW для владельца и
                           # R для группы в всех остальных
                           # пользователей
chmod 0777, 'script.pl'    # Назначает права RWX для всех
                           # пользователей (не очень грамотное решение)
chmod 0000, 'cia.dat'      # Запрещает доступ к файлу со стороны
                           # любого пользователя
```

Выше на этом занятии мы рассматривали функцию **mkdir**. При описании второго аргумента этой функции мы говорили, что он задает права доступа к каталогу (наподобие того, как функция **chmod** задает права доступа к файлу). Вот несколько примеров:

```
mkdir "/usr/tmp", 0777   # Создается каталог, полностью открытый
                         # для всех пользователей
mkdir "myfiles", 0700    # Создается каталог, открытый только для
                         # его владельца
```



Права доступа к файлам в **UNIX** часто называют режимом доступа (mode). Поэтому название команды **chmod** расшифровывается как *change mode* (изменить режим доступа).

Получение информации о файле

ДЛЯ получения подробной информации о файле в Perl используется функция stat. Поскольку прототип этой функции взят из UNIX, а структура файловой системы UNIX отличается от принятой в других ОС, возвращаемые функцией stat данные зависят от конкретной операционной системы. Синтаксис функции stat выглядит так:

```
stat дескриптор_файла;
stat имя_файла;
```

Таким образом, функция stat позволяет получать информацию как по дескриптору файла (если файл уже был открыт), так и по имени файла. В любой операционной системе функция stat возвращает список, состоящий из 13 элементов, которые описывают атрибуты файла. Сами значения элементов этого списка зависят от типа операционной системы, в которой используется интерпретатор Perl. Причина заключается в том, что файловые системы некоторых операционных систем могут не поддерживать те или иные атрибуты. В табл. 10.3 описаны возвращаемые функцией stat элементы списка для двух популярных операционных систем: UNIX и Windows.

Таблица 10.3. Описание возвращаемых функцией stat значений

Номер элемента	Название	UNIX	Windows
0	dev	Номер устройства	Номер диска (обычно 2 — это диск C:, 3 — D: и т.д.)
1	ino	Число индексных дескрипторов (inode)	Всегда нуль
2	mode	Режим доступа к файлу	Не определено
3	nlink	Число ссылок (links) на файл	Обычно 0, хотя в системе NTFS допускается наличие ссылок на файл
4	uid	Идентификатор владельца файла (user id)	Всегда нуль
5	gid	Идентификатор группы (group id)	Всегда нуль
6	rdev	Специальная информация о файле	Номер диска (повторно)
7	size	Размер файла в байтах	Размер файла в байтах
8	atime	Время последнего обращения к файлу	Время последнего обращения к файлу
9	mtime	Время последней модификации файла	Время последней модификации файла
10	ctime	Время последней модификации индексного дескриптора	Время создания файла
11	blksz	Размер блока на диске	Всегда нуль
12	blocks	Количество блоков, выделенных для файла	Всегда нуль

Большинство элементов, описанных в табл. 10.3, вы никогда не будете использовать, однако они приведены для полноты изложения. Чтобы получить более подробную информацию об остальных элементах (особенно относящихся к системе UNIX), обратитесь к руководству по операционной системе.

Вот пример использования функции stat:

```
@stuff=stat "myfile";
```

Обычно возвращаемые функцией stat значения присваивают списку скаляров, которым назначены осмысленные имена, например:

```
($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size,
 $atime, $mtime, $ctime, $blksize, $blocks)=stat("myfile");
```

А теперь давайте распечатаем код прав доступа к файлу в виде восьмеричного числа, о котором шла речь выше, в разделе "Немного о правах доступа к файлам":

```
printf "%04o\n", $mode&0777;
```

В этом фрагменте кода некоторые элементы вам могут показаться новыми. Ничего удивительного, о них мы до сих пор еще не упоминали. В информации о правах доступа, возвращаемой функцией stat (и в данном примере помещаемой в переменную $mode), содержатся "лишние" данные. Поэтому используется конструкция &0777, которая отбрасывает "лишнюю" информацию и оставляет только то, что нам нужно. Данная операция называется *маскированием* данных, или наложением *маски,* а число 0777 называется *маской.* И, наконец, шаблон %04o в функции printf задает восьмеричный формат представления чисел (используются только цифры от 0 до 7), в котором принято в системе UNIX отображать коды прав доступа. Цифра 4 в шаблоне задает ширину поля, а 0 перед ней говорит о том, что к трем восьмеричным числам будет добавлен незначащий нуль.



В *восьмеричном* формате числа представляются точно так же, как и в десятичном, но по основанию 8. При этом значение каждого разряда восьмеричного числа может быть в пределах от 0 до 7. При прибавлении к 7 единицы возникает переполнение. При этом младший разряд сбрасывается в нуль, а к старшему прибавляется единица. Использование восьмеричного формата представления чисел в UNIX, главным образом, дань традиции, которая перешла и в Perl. Если вы окончательно сбиты с толку, не паникуйте. Просто используйте приведенный выше шаблон функции printf для отображения кодов прав доступа к файлам и ни о чем не думайте. Восьмеричная система счисления используется не так часто, поэтому не нагружайте себя лишней информацией.

В табл. 10.3 упоминается о трех моментах времени, связанных с файлом. Речь идет о времени доступа к файлу, времени его модификации и времени изменения индексного дескриптора (или создания файла). Здесь под моментом времени следует понимать точную дату и время выполнения операции, которая хранится в довольно необычном формате. В Perl отсчет времени производится в секундах, прошедших с 0 часов по Гринвичу 1 января 1970 года. Поэтому, чтобы вывести дату и время в привычном нам формате, следует воспользоваться функцией localtime, как показано ниже на примере:

```
print scalar localtime($mtime);
```

Этот оператор выводит дату модификации файла в формате, подобном Mon May 15 14:44:55 2000. Время доступа к файлу (поле $atime) — это момент времени, когда файл был последний раз открыт для чтения или записи, а время модификации (поле $mtime) — когда содержимое файла было последний раз изменено. В системе UNIX в поле $ctime отмечаются моменты времени, когда изменяется индексный дескриптор файла. А это происходит при изменении владельца файла, прав доступа к нему, числа ссылок на файл и т.д. Таким образом, не стоит полагаться на то, что в данном поле будет находиться время создания файла, хотя в большинстве случаев так оно и есть. В системе Windows в поле $ctime находится время создания файла.

Иногда для работы необходимо только одно значение из списка, возвращаемого функций stat. В таком случае поместите вызов функции stat в круглые скобки и для выделения нужного элемента из временного списка укажите его номер в квадратных скобках, как показано ниже.

```
print "Размер файла: ",(stat("file"))[7], " байтов";
```

Упражнение: переименование группы файлов

ВЫПОЛНИВ данное упражнение, вы создадите еще один полезный инструмент, который облегчит вашу дальнейшую работу. Эта небольшая программа позволяет выполнить переименование группы файлов по заданному шаблону, находящихся в указанном каталоге. Предположим, что в каталоге находятся файлы `Chapter_01.rtf`, `Chapter_02.rtf`, `Chapter_04.rtf` и т.д., которые мы хотим переименовать в `Hour_01.rtf`, `Hour_02.rtf`, `Hour_04.rtf` и т.д. Выполнить эту задачу средствами командной оболочки дело не из легких, не говоря уже о графических файловых оболочках наподобие программы Проводник в Windows.

С помощью текстового редактора введите программу, исходный код которой приведен в листинге 10.3, и сохраните ее в файле `Renamer`. Сделайте файл выполняемым, как это было описано на 1-м занятии, "Начало работы с Perl".

Завершив подготовительные действия, запустите программу с помощью следующей команды:

```
perl Renamer
```

В листинге 10.4 приведен пример диалога с программой.

Листинг 10.3. Исходный текст программы Renamer

```
#!/usr/bin/perl

use strict;

my($dir, $oldpat, $newpat);
print "Укажете каталог: ";
 chomp($dir=<STDIN>);
print "Введите шаблон для поиска файлов: ";
 chomp($oldpat=<STDIN>);
print "Введете шаблон для переименоваввя файлов: ";
chomp($newpat=<STDIN>);

opendir(DH, $dir) || die "Ошибка при открытии каталога $dir: $!";
my @files=readdir DH;
close(DH);
my $oldname;
 foreach(@files){
     $oldname=$_;
    s/$oldpat/$newpat/;
    next if (-e "$dir/$_");
   if (! rename "$dir/$oldname", "$dir/$_") {
    warn "Ошибка при переименовании файла $oldname в $_: $!";
   } else {
        print "Файл $oldname переименовав в $_\п";
   >
}
```

 Проведем анализ программы.

- *Строки 13-15.* Все элементы каталога $dir помешаются в массив @files.
- *Строки 17—19.* Выполняется цикл по всем элементам массива @files. Очередной элемент массива помешается в переменную $_, а затем присваивается переменной $oldname. Затем в строке 19 исходное имя файла заменяется в переменной $_ на новое.
- *Строка 20.* Перед переименованием необходимо убедиться, что файла с таким именем нет в каталоге. В противном случае после переименования первоначальный файл будет утерян.
- *Строки 21—25.* Выполняется переименование файла. Если по какой-либо причине эта операция не может быть выполнена, выводится соответствующее сообщение. Обратите внимание, что перед именами файлов необходимо указать каталог, например $dir/$oldname. Причина заключается в том, что в массиве @files и переменной $_ находятся только имена файлов без пути.

Листинг 10.4. Результат работы программы Renamer

```
Укажите каталог: tmp
Введите шаблон для поиска файлов: Chapter
Введите шаблон для переименования файлов: Hour
Файл Chapter_4.rtf переименовав в Hour_4.rtf
Файл Chapter_2.rtf переименован в Hour_2.rtf
Файл Chapter_1.rtf переименован в Hour_1.rtf
```

Резюме

На этом занятии речь шла о создании, удалении и переименовании каталогов с помощью функций Perl rakdir, rmdir и rename. Кроме того, вы узнали, как получить дополнительную информацию о файле (размер, время последнего обращения и др.) с помощью функции stat. В качестве примеров на этом занятии были созданы две простые, но очень полезные программы, которые позволяют автоматизировать рутинную работу.

Вопросы и ответы

Почему приведенный ниже фрагмент программы не работает? Несмотря на то что в каталоге есть файлы, содержимое каталога не читается.

```
opendir(DIRHANDLE, "/mydir") || die;
@files=<DIRHANDLE>;
closedir(DIRHANDLE);
```

Проблема заключается во второй строке кода. Переменная DIRHANDLE является дескриптором каталога, а не файла! Содержимое каталога нельзя читать с помощью углового оператора (о). Чтобы исправить ошибку, замените вторую строку на @files=readdir DIRHANDLE

Почему функция glob("*.*") возвращает не все файлы, находящиеся в каталоге?

Потому что шаблон *.* соответствует только тем файлам, в имени которых содержится точка. Чтобы получить список всех файлов каталога, используйте конструкцию glob("*"). Функция glob и ее шаблоны реализованы в Perl так, чтобы обеспечить переносимость программ между разными компьютерными платформами. Поэтому шаблон *.* не соответствует его аналогу в MS-DOS.

Для выполнения поиска файлов во вложенных каталогах в программу пудгер была добавлена функция opendir и несколько модифицирован основной цикл. Однако в программе появились какие-то ошибки.

Если честно — вам не нужно с нуля писать такую программу. Задача обхода дерева каталогов уже давным-давно решена. И хотя она не из легких, существует множество ее решений. Поэтому незачем изобретать велосипед. Если же вы хотите просто попрактиковаться, обратитесь к материалу 15-го занятия, "Обработка данных в Perl". На нем рассмотрен модуль File::Find, облегчающий решение описанной задачи и, что более важно, ее последующую отладку.

Почему программа, описанная в листинге 10.3, выдает сообщение об ошибке при попытке переименования файлов *.bat в *.tmp?

Дело в том, что в качестве шаблона для поиска файлов нельзя использовать конструкцию *.bat, поскольку она не является корректным регулярным выражением. Звездочка должна обязательно следовать после какого-нибудь символа шаблона, ведь она обозначает факт повтора предыдущего символа. Если же ввести конструкцию *\.bat, сообщение об ошибке исчезнет, но программа все равно работать не будет, поскольку вряд ли в вашем каталоге будет находиться файл, начинающийся с "*.bat". И вообще файлы, имена которых начинаются со звездочки, — скорее исключение, чем правило.

Для решения проблемы либо введите корректный шаблон, либо измените строку19 программы следующим образом:

```
s/\Q$oldpat/$newpat/;
```

В результате будет запрещено использование специальных символов в регулярных выражениях, и они будут восприниматься как обычные литералы.

Семинар

Контрольные вопросы

1. Какой оператор используется для вывода времени последнего изменения файла foofile?

   а) print glob("foofile");

   б) print (stat("foofile"))[9];

   в) print scalar localtime (stat{"foofile"))[9].

2. Что возвращает функция unlink?

   а) число удаленных файлов;

   б) истинное или ложное значение в зависимости от результата выполнения;

   в) число файлов, которые должны были быть удалены.

Ответы

1. Правильный ответ — вариант б) или в). В первом случае время модификации файла будет выведено в секундах, прошедших с 0 часов 1 января 1970 года, что мало информативно. Во втором случае дата и время будут выведены в удобном для пользователя формате.

2. Правильным является вариант а), хотя вариант б) с некоторой натяжкой можно также назвать правильным. Если функция unlink не сможет удалить ни одного файла, она возвратит 0, что соответствует ложному значению.

Упражнения

- Попытайтесь в качестве упражнения (*и только.*) написать программу, которая бы выводила список файлов текущего и всех вложенных каталогов.

11-й час

Взаимодействие с операционной системой

До сих пор при изучении Perl мы не выходили за рамки этого языка, так как в нем были предусмотрены все необходимые средства: сортировка данных, получение списка файлов каталога, обработка файлов конфигурации и др. Однако довольно часто возникают задачи, для решения которых требуется привлечение внешних программ. Здесь нет ничего удивительного, поскольку нельзя объять необъятное и предусмотреть средства на все случаи жизни.

Про Perl часто говорят, что он является великолепным *интегрирующим языком (glue language)*. А это означает, что в программах на Perl можно воспользоваться другими приложениями, которые являются компонентами операционной системы, объединить их вместе и создать на их основе другое (более функииональное) приложение. С помощью Perl можно запустить любую утилиту операционной системы, передать ей данные, получить результат их обработки, а затем завершить ее работу.

Perl позволяет объединить несколько простых и малофункциональных утилит в одно полезное приложение. В результате существенно повышается скорость разработки программ и, что самое важное, не тратится драгоценное время на их отладку. Программист получает возможность использовать удобные ему средства разработки, что в конечном счете повышает скорость и качество разработки. Таким образом, интеграция системных утилит сулит значительные преимущества.

Основные темы этого занятия.

- Использование функции `system()`.
- Перенаправление выходного потока.
- Проблема переносимости программ.



Большинство примеров данного занятия состоит из двух частей. Одна часть предназначена для выполнения в среде Windows и DOS, а другая — в среде UNIX. Если же будет приведен только один пример, то мы обязательно укажем, какие изменения (обычно небольшие) нужно внести в программу, чтобы она работала в той или иной системе.

Функция system()

Простейший способ запустить команду операционной системы из Pert — воспользоваться функцией system(). Эта функция приостанавливает текущую программу на Perl, выполняет внешнюю программу, после чего возобновляет выполнение программы на Perl. Синтаксис функции system{) выглядит так:

system *команда;*

Здесь вместо параметра *команда* нужно указать имя программы, которую необходимо выполнить. Если внешняя программа завершилась успешно, функция system() возвращает код 0. Если же в процессе выполнения внешней программы произошло что-либо непредвиденное, возвращается ненулевой код возврата. Обратите внимание, что эти значения согласуются с точностью "до наоборот" со значениями true и false, принятыми в Perl.

А теперь рассмотрим пример функции system() для UNIX.

```
system("ls -lF"); # Вывести содержимое каталога
# Распечатаем документацию на функцию system
if ( system("perldoc -f system") ) {
    print "Документация к Perl не установлена!\n";
}
```

А вот как будет выглядеть тот же самый пример для системы Windows:

```
system("dir /w"); # Вывести содержимое каталога
# Распечатаем документацию на функцию system
if ( system("perldoc -f system") ) {
    print "Документация к Perl не установлена!\n";
}
```

Как видите, функция system!) работает одинаково в обеих системах. Однако не стоит забывать, что рассматриваемые нами операционные системы имеют совершенно разный набор команд. Так, для получения содержимого каталога в системе DOS используется команда dir, а в UNIX — ls. В то же время команда perldoc не зависит от типа операционной системы и работает совершенно одинаково как в DOS, так и в UNIX. Но такое случается довольно редко.

После запуска внешней программы с помощью функции system() все ее сообщения выводятся на терминал, так же, как и сообщения программы на Perl. Если же для продолжения вычислений внешней программе понадобятся исходные данные, их можно будет ввести с терминала точно так же, как и данные для программы на Perl. При запуске внешней программы она наследует стандартные потоки ввода и вывода STDIN и STDOUT вызвавшей ее программы на Perl. Таким образом, операции ввода/вывода для внешней программы будут выполняться над теми же источниками данных. Таким образом, с помощью функции system() можно без проблем вызывать внешние интерактивные программы.

Рассмотрим пример для системы UNIX.

```
$file="myfile.txt";
system{"vi $file");
```

Для систем Windows и DOS этот пример будет выглядеть так:

```
$file="myfile.txt";
system("edit $file");
```

В каждом из представленных фрагментов выполняется запуск текстового редактора *для* внесения изменений в файл myfile.txt. Для системы UNIX таким редактором является vi, а для DOS — edit. Текстовый редактор будет выполняться в полноэкранном режиме и, естественно, все его внутренние клавиатурные команды будут работать как обычно. По завершении работы с редактором управление возвращается программе на Perl.

Функцию system() можно использовать для запуска любых программ, а не только консольных приложений (которые работают в текстовом режиме) операционной системы. Например, в системе UNIX для запуска приложения, отображающего часы в графическом режиме, используется следующий оператор Perl:

```
system("xclock -update 1");
```

Чтобы запустить редактор Notepad в Windows, воспользуйтесь следующим оператором Perl:

```
system("notepad.exe myfile.txt");
```

Использование средств командной оболочки

Функция system(), как и большинство других функций, рассмотренных на данном занятии, позволяет воспользоваться всеми преимуществами командной оболочки операционной системы, в которой вы работаете. Так происходит потому, что перед вызовом внешней программы функция system() запускает копию командной оболочки (в UNIX это /bin/sh, а в Windows и DOS — command.com), которая и выполняет эту программу. Как следствие, при вызове внешней команды вы можете перенаправлять потоки ввода/вывода (>), выполнять конвейерную обработку (|), запускать задания в фоновом режиме в системе UNIX (&), а также пользоваться любыми доступными средствами оболочки.

Например, чтобы запустить внешнюю программу и перенаправить ее вывод в файл, используется следующий оператор Perl:

```
system("perldoc perlfaq5 > faqfile.txt");
```

Эта функция передает системной оболочке команду perldoc perlfaqS и перенаправляет стандартный выходной поток в файл faqfile.txt. Заметьте, что синтаксис данного оператора одинаков как для UNIX, так и для DOS.

Как и следовало ожидать, некоторые возможности, такие как перевод задачи в фоновый режим, использование регулярных выражений и др., работают только в системе UNIX, поскольку в командной оболочке DOS и Windows они не поддерживаются. Вот пример:

```
# Отсортируем файл, имя которого задано в переменной $f,
# и распечатаем результат
system("sort $f | lp"); # В некоторых системах для печати
                        # используется команда lpr
# Запустим программу Xterm в фоновом режиме
system("xterm &");
```

В последнем примере после запуска программы xterm управление будет сразу же передано программе на Perl, поскольку символ амперсанда (&), расположенный после имени команды, предписывает оболочке запустить программу xterm в *фоновом режиме*. При этом интерпретатор Perl не будет ожидать завершения ее работы.



В системе UNIX для запуска внешних программ с помощью функции `system()`, выполнения конвейерной обработки и подстановок команд (об этом речь пойдет чуть ниже) интерпретатор Perl использует оболочку /bin/sh или ее аналог. Причем этот процесс не зависит от типа оболочки, в которой работает пользователь. Так сделано для того, чтобы обеспечить максимальную переносимость программ на разные платформы UNIX.

Некоторые из рассмотренных в данной главе примеров с использованием функции `system()` не будут работать на компьютерах Macintosh. Особенно это относится к подстановкам команд и конвейерной обработке. За дополнительной информацией обратитесь к разделу Macintosh Specific Features справочной системы MacPerl.

Перенаправление выходного потока

Несмотря на все преимущества, у функции `system()` есть также один существенный недостаток. Она не позволяет перенаправить выходной поток программы на дальнейшую обработку интерпретатору Perl. Чтобы устранить описанный недостаток, можно воспользоваться обходным маневром, как показано ниже на примере.

```
I Учтите, что команды 'dir' и 'ls' используются здесь
# только для примера! В большинстве случаев намного эффективнее
# будут работать функции opendir/readdir/closedir
system("dir > outfile"); # В UNIX вместо 'dir' используйте команду 'ls'
open(OF, "outfile"} || die "Ошибка при открытии файла outfile: $1";
@data=<OF>;
close(OF);
```

В этом примере выходные данные команды dir (или `ls`) перенаправляются командной оболочкой в файл outfile. Затем этот файл открывается и его содержимое помещается в массив `@data`. Таким образом, в этом массиве будут содержаться данные, полученные от команды dir.

Описанный метод слишком громоздкий и не вполне очевидный. Поэтому нет ничего удивительного в том, что в Perl предусмотрено более элегантное решение — использование оператора *подстановки команд,* или *обратных кавычек*. Любая команда в Perl, заключенная в обратные кавычки (`` ` ` ``), рассматривается как внешняя программа и запускается на выполнение по аналогии с функцией `system()`. При этом выходной поток внешней программы перехватывается и возвращается в вызывающую программу так же, как это происходит при вызове функций, например:

```
$directory=`dir`; # 8 UNIX вместо 'dir' используйте команду 'ls'
```

В этом примере данные, полученные в результате выполнения команды dir, помещаются в переменную $directory.

Как и в функции `system()`, внутри обратных кавычек можно пользоваться всеми доступными средствами командной оболочки: символ > вызывает перенаправление выходного потока, символ | выполняет конвейерную обработку и (в системе UNIX) символ s позволяет запустить программу в фоновом режиме. Только не забудьте, что при перенаправлении выходного потока программы или ее запуске в фоновом режиме в программу на Perl не возвращаются никакие данные.

В скалярном контексте оператор подстановки команд возвращает выходные данные программы в виде текстовой строки. Если в данных содержится несколько строк текста, они разделяются в строке специальным символом — разделителем за-

писей. В контексте списка выходные данные программы построчно присваиваются переменным списка. При этом в конце каждой строки помещается разделитель записей. Вот пример:

```
@dir=`dir`; # В UNIX вместо 'dir' используйте команду 'ls'
foreach (@dir) {
# Обработка каждой строки в отдельности
}
```

Здесь в цикле foreach обрабатывается каждая строка, находящаяся в массиве @dir.

В Perl существует альтернативная форма записи оператора подстановки команд. Вместо обратных кавычек можно использовать оператор qx{>. Команду, которую нужно выполнить, следует поместить в фигурные скобки, как показано ниже:

```
$perldoc=qx{perldoc perl};
```

Оператор qx позволяет повысить читабельность программы, если при записи внешней команды внутрь обратных кавычек следует поместить служебные символы, такие как прямые и обратные кавычки, косую черту и др. (Напомним, что перед служебными символами следует поставить обратную косую черту.) Вот пример.

```
$complex=`sort \`grep -l 'conf' *\``; # Выглядит не очень понятно
```

А теперь перепишем этот оператор так:

```
$complex=qx{ sort `grep -1 'conf *`};# Намного яснее
```

Вместо фигурных скобок можно использовать произвольные символы, а также любые парные символы, такие как о, () и [ ].

Как избежать обращения к командной оболочке

В некоторых случаях бывает трудно определить границы области действия интерпретатора Perl и командной оболочки. Давайте рассмотрим два примера.

Для UNIX:

```
$myhome=*ls $HOME`;
```

Или то же самое для DOS и Windows:

```
$myhome=`dir %windir%`;
```

Как определить, к чему в первом примере относится переменная $HOME? Является ли она переменной Perl, или переменной окружения командной оболочки? А во втором примере? Является ли конструкция %windir% переменной окружения командного интерпретатора command.com, или это хэш языка Perl, за которым помещен знак процента?

Хуже всего то, что переменная $HOME интерпретируется Perl. А это означает, что она является скаляром Perl, а не переменной окружения командной оболочки, как вы, вероятно, предполагали. Таким образом, внутри обратных кавычек переменные заменяются их значениями, точно так же, как это происходит и внутри двойных кавычек (" ")• Однако данное правило не относится к хэшу — только к массивам и скалярам. Таким образом, во втором примере конструкция %windir% относится к переменной окружения командного интерпретатора command.com.

Чтобы избежать интерпретации переменных внутри обратных кавычек, поместите перед ними символ обратной косой черты, как показано в следующем примере:

```
$myhome=`ls \$HOME`; # Символ 'V прикрывает переменную $HOME
```

А вот пример для DOS и Windows:

```
$windows=`dir %windir%`;
```

В этих примерах используется значение переменной окружения HOME в UNIX и windir в DOS.

При использовании альтернативной формы записи оператора подстановки команд в конструкции qx{} следует заменить **символ-ограничитель**, как показано в следующем примере:

```
$myhome=qx' ls $HOME ';
```

Или для Windows и DOS:

```
$windows=qx' dir %windir% ';
```

Конструкция qx'' распознается интерпретатором Perl и обрабатывается особым образом: внутри одинарных кавычек не выполняется замена переменных на их значение. Таким образом, в операторе подстановки команд появляется возможность использовать любые служебные символы, не помещая перед ними символ обратной косой.

Конвейерная обработка

Конвейерная обработка используется в системах UNIX и DOS/Windows для передачи данных между процессами. Она позволяет связать выходной поток одного процесса с входным потоком другого. Давайте рассмотрим следующий набор команд DOS, который с небольшими изменениями (команду dir нужно заменить на ls) будет работать и в UNIX:

```
dir > outfile
sort outfile > newfile
more newfile
```

Здесь выходные данные команды dir перенаправляются в файл outfile. Затем содержимое этого файла сортируется с помощью команды sort и записывается в новый файл newfile. И, наконец, содержимое файла newfile поэкранно выводится на терминал.

Конвейерная обработка позволяет выполнить те же самые действия, но без привлечения дополнительных временных файлов outfile и newfile:

```
dir | sort j more
```

В этом примере выходные данные команды dir подаются на вход команды sort, которая выполняет их сортировку. Затем отсортированные данные подаются на вход команды more для поэкранного отображения. При этом не требуется перенаправлять выходной поток (>) вq временный файл, поскольку операционная система сделает все сама!

Выше был приведен пример *конвейерной* обработки данных, признаком которой служит вертикальная черта. Конвейеры активно используются в системе UNIX для обмена данными между многочисленными утилитами операционной системы. В DOS и Windows конвейерная обработка также поддерживается, но используется сравнительно редко и постепенно переходит в небытие по мере вытеснения текстовых утилит программами с графическим пользовательским интерфейсом.

Программы на Perl также могут участвовать в конвейерной обработке данных. Давайте создадим одну из таких программ, которая будет читать данные из стандартного входного потока, выполнять над ними определенные действия, а затем выводить в выходной поток, как показано в следующем примере:

```
dir /B | sort | perl Totaler | more
```

В этом конвейере программа Totaler написана на Perl (листинг 11.1). Она выполняет подсчет количества файлов в каталоге и их суммарный размер. В системе UNIX замените команду dir /B на ls -1.

Листинг 11.1. Исходный текст программы Totaler

```
1:  #!/usr/bin/perl
2:
3:  use strict;
4:  my($dirs,$sizes,$total);
5:
6:  while(<STDIN>){
7:      chomp;
8:      $total++;
9:      if (-d $_) {
10:         $dirs++;
11:         print "$_\n";
12:         next;
13:     }
14:     $sizes+=(stat($_))[7];
15:     print "$_\n";
16: }
17: print "Всего файлов - $total, каталогов - $dirs\n";
18: print "Суммарный размер файлов - ", $sizes/($total-$dirs), "\n";
```

Проведем анализ программы.

- *Строка 6.* В цикле выполняется чтение строк из стандартного входного потока. При этом каждая строка присваивается переменной $_. При потоковой обработке дескриптору STDIN текущей программы соответствует дескриптор STDOUT предыдущей. Таким образом, в нашем примере из дескриптора STDIN считываются данные, полученные в результате выполнения команды dir /B.
- *Строки 9—13.* Если встретился каталог, увеличим счетчик каталогов, находящийся в переменной Sdirs. При этом имя каталога распечатывается и цикл повторятся снова.
- *Строки 14—15.* В противном случае размер файла прибавляется к содержимому переменной $sizes и имя файла распечатывается.
- *Строки 17—18.* Накопленные статистические данные (количество файлов, каталогов и общий размер файлов) выводятся на печать.

Еще один способ конвейерной обработки данных заключается в том, что конвейер можно рассматривать как файл, информация в который может записываться, а затем считываться. Открыть такой файл можно с помощью функции Perl open, как показано ниже:

```
# В системе UNIX замените команду 'dir /B' 'ls -1'
open(RHANDLE, "dir /в| sort |"J ||
        die "Ошибка при открытие конвейера для чтения: $!";
```

В этом примере функция open открывает конвейер для чтения данных, полученных в результате выполнения цепочки команд dir /B | sort. Вертикальная черта, расположенная в цепочке команд крайней справа, говорит о том, что конвейер открывается для чтения. При выполнении функции open Perl запускает цепочку команд dir /B | sort и помещает полученные от команды sort данные во временный файл. Поэтому при чтении дескриптора RHANDLE эти данные попадут в программу на Perl для дальнейшей обработки.

А теперь рассмотрим еще один пример:

```
open(WHANDLE, "| more") ||
        die "Ошибка при открытии конвейера для записи: $!";
```

В этом случае функция open открывает конвейер для записи данных, которые подаются на вход команды more. Вертикальная черта, расположенная в цепочке команд крайней слева, говорит о том, что конвейер открывается для записи. Таким образом, все данные, выведенные в дескриптор WHANDLE, будут переданы в буфер программы more для поэкранного отображения. Кстати, вот вам один из способов, как можно заставить программу отображать данные постранично.

После того как будут закончены все операции чтения/записи с дескрипторами наподобие RHANDLE и WHANDLE, нужно обязательно закрыть эти дескрипторы и соответствующие им конвейеры. Тогда сможет корректно завершиться программа, запушенная функцией open. Если же после окончания работы с конвейером оставить дескриптор открытым, то внешняя программа останется в подвешенном состоянии после того, как программа на Perl завершит свою работу.

При закрытии конвейера с помощью функции close по коду возврата можно судить о том, как была выполнена конвейерная обработка. Поэтому всегда проверяйте, успешно или нет был закрыт конвейер, как показано ниже:

```
close(WHANDLE) |} warn "Ошибка при закрытии конвейера: $!";
```



При изучении материала этого раздела у вас мог возникнуть вопрос; "Почему во время открытия конвейера с помощью функции open по коду возврата нельзя судить о том, успешно или нет была выполнена обработка данных?" Все дело в архитектуре системы UNIX. После того как интерпретатор Perl создал конвейер и выдал команду на его запуск, операционная система не обязательно сразу его запустит на выполнение. Поэтому остается надеяться, что если конвейер создан правильно и успешно запущен, то он корректно и завершит свою работу. После того как последняя программа в цепочке закончит свою работу, она должна вернуть соответствующий код возврата. Функция close считывает этот код, анализирует его, принимает решение о том, успешно или нет была выполнена обработка данных, и возвращает соответствующий код возврата.

Общие сведения о переносимости программ

Переносимость — это одно из основных преимуществ Perl, благодаря которому он и завоевал столь широкую популярность- Интерпретатор Perl гарантирует, что программа практически без изменений будет работать одинаково на любой из поддерживаемых

компьютерных платформ (VMS, UNIX, Macintosh или DOS). Если в программе используются средства взаимодействия с операционной системой, такие как операции ввода/вывода, интерпретатор Perl пытается самостоятельно выполнить всю черновую работу и скрыть от программиста детали, обеспечивая при этом работоспособность кода. Однако в некоторых редких случаях требуется вмешательство программиста.



На 16-м занятии, "Сообщество РегГ, мы обсудим более подробно причины столь высокой переносимости программ на Perl.

На этом занятии мы уже не раз отмечали, что некоторые участки программ будут работать только в Windows и DOS, а другие — только в UNIX. Поэтому при написании программы вы должны учитывать тип операционной системы, на которой ее предполагается запускать. Это типичный пример *машинно-зависимого* программирования. При таком подходе требуется создать отдельные версии программ для каждой из операционных систем, например для Windows и UNIX. Наличие нескольких версий программы создает проблемы при разработке версии программы для третьей операционной системы, например MacOS 9.

Однако сказанное выше вовсе не означает, что профаммы на Perl, написанные для одной компьютерной платформы, например Windows NT, не будут работать на другой, например UNIX. Более того, чаще всего именно так и происходит — профаммист работает в одной системе, а пользователи профаммы в другой. У некоторых пользователей даже складывается такое впечатление, что, поскольку интерпретатор Perl создан для большинства современных компьютерных платформ, выполнение программы в среде Windows NT ничем не отличается от среды UNIX. Переносимые профаммы имеет смысл создавать только в том случае, если они предназначены для работы на различных компьютерных платформах, как, например, Web-серверы и вспомогательные профаммы для них.

Учтите, что создание различных версий одной и той же профаммы, предназначенных для работы на всех возможных компьютерных платформах и во всех возможных ситуациях, — занятие очень хлопотное, непродуктивное и малоприятное. Ниже приведено несколько правил, выполнение которых гарантирует, что ваша профамма без изменений (или с незначительными изменениями) будет работать на любой платформе.

- Всегда включайте режим выдачи предупреждений и используйте директиву use strict. Это позволит гарантировать, что ваш код будет корректно выполняться различными интерпретаторами Perl и что в профамме не будет фубых ошибок, о которых может предупредить компилятор.
- Всегда проверяйте коды возврата после вызова системных функций. Например, при открытии файла используйте конструкцию open || die. Никогда не используйте оператор open сам по себе. Проверка кодов возврата позволяет выявить ошибки при переносе профамм с одного сервера на другой, а не только от одной платформы на другую.
- Выводите краткие, но понятные сообщения об ошибках.
- По возможности отдавайте приоритет встроенным функциям Perl перед функцией system и помещением вызова внешней профаммы в обратные кавычки.
- Создавайте для всех системно зависимых операций (ввод/вывод, управление процессами и др.) соответствующие функции на Perl, которые будут играть роль оболочки. Не забудьте также проверить, что используемые вами средства реализованы в текущей операционной системе.

С первыми двумя рекомендациями вы уже должны быть хорошо знакомы. Во всех примерах этой книги анализируется код возврата после выполнения критичных функций, а начиная с 8-го занятия, "Функции", во всех больших примерах используется директива use strict и режим выдачи предупреждений.

С третьей рекомендацией о выводе понятных сообщений об ошибках трудно не согласиться. В самом деле, что может быть понятнее сообщений, приведенных ниже?

```
(no message, or wrong output)
Died at line 15.
Cannot open Foofile.txt: No such file or directory
Cannot open Foofile.txt: No such file or directory at myscript.pl line 24
```

Очевидно, что последняя строка наиболее информативна. И даже если с момента написания программы пройдет достаточно много времени, вы всегда сможете определить место в программе (файл myscript.pl, строка 24), где произошла ошибка, и причину (не найден файл Foofile.txt) сбоя. Такая подробная информация поможет быстро устранить проблему. При написании программы не жалейте нескольких минут времени на создание хороших, информативных сообщений об ошибке. В будущем оно окупится сторицей.

Четвертая рекомендация означает, что везде, где только возможно, нужно использовать встроенные средства Perl. Например, для получения списка файлов каталога проще всего воспользоваться оператором наподобие `$dir=`dir``. Однако он не будет работать на платформах, отличных от Windows. Поэтому лучше воспользоваться конструкцией <*> либо набором • функций opendir/readdir/closedir (что предпочтительнее). При таком подходе ваша программа будет работать на любой платформе.

Как быть с **отличиями?**

Последняя рекомендация по написанию переносимых программ, рассмотренная в предыдущем разделе, требует небольшого пояснения на примерах. Напомним, что речь идет об использовании функций-оболочек для системно зависимых вызовов.

При написании программ на Perl не стоит забывать, что рано или поздно настанет момент, когда их придется запускать в другой операционной системе или на компьютере другого типа. А вдруг вы создадите нечто наподобие Web-сервера amazon.com, и ваш IBM PC не будет справляться с нахлынувшим потоком посетителей? Тогда вам придется перенести свой сервер на более мощный компьютер под управлением Windows NT или UNIX, или даже поместить его в кластер, состоящий из 10000 серверов UNIX на базе Sun Enterprise. Или другой более реалистичный пример. Предположим, что вы пользуетесь некоторым CGI-сценарием и решили поменять Web-провайдера, поскольку новый провайдер предоставляет более широкий набор услуг. Подобная ситуация случается сплошь и рядом и не требует особых комментариев.

Итак, как же программа сможет определить, на какой платформе она выполняется, и учесть отличия между Windows и UNIX? Все очень просто. В Perl предусмотрена специальная переменная $^O (знак доллара, за которым следуют символ вставки и прописная буква *O),* содержащая имя архитектуры компьютерной платформы, на которой выполняется программа. Например, в среде Windows и DOS ей присвоена строка MSWin32. В UNIX с помощью данной переменной можно определить тип операционной системы: linux, aix, solaris, freebsd и т.д.

Ниже перечислено несколько типичных системно зависимых задач, которые часто приходится решать в пользовательских программах.

- Поиск какой-либо информации, относящейся к конфигурации операционной системы.

- Работа с диском и структурой каталогов.
- Использование системных служб, например отправка электронной почты.

В качестве примера рассмотрим программу, с помощью которой можно легко определить количество свободного дискового пространства в системе. Она будет полезна в случае, если один из пользователей захочет загрузить из Internet какой-либо файл и перед загрузкой проверить, поместится ли он на локальном диске. Фрагмент программы на Perl, оценивающий в системе Windows количество свободного дискового пространства для устройства, на котором находится текущий каталог, выглядит так:

```
# В последней строке сообщения команды 'dir' должно находиться нечто
# наподобие: 10 dir(s)      67,502,080 bytes free
# В системе Windows 98 вместо слова 'bytes' может выводиться 'MB'
my(@dir,$free);
@dir=`dir`;
$free=$dir[$#dir];
$free=~s/.*([\d,]+) \w+ free/$1/;
$free=~s/,//g;
```

В этом фрагменте кода выбирается последняя строка листинга, полученного с помощью команды dir и помещенного в массив @dir. Далее с помощью регулярных выражений из этой строки выделяются числа и запятые, расположенные перед фразой bytes free. И, наконец, с помощью еше одного регулярного выражения из числа удаляются запятые, которые служат в качестве разделителей тысяч. В результате в переменной $free будет находиться размер свободного места на диске в байтах. Рассмотренный нами пример прекрасно работает в системе Windows. Для UNIX следует воспользоваться таким фрагментом программы:

```
# В последних строках сообщения команды 'df -k'
# должно находиться нечто наподобие:
#
# Filesystem  1K-blocks    Used    Avail  Capacity  Mounted on
# /dev/wd0s1a     31775   21431     7802     73%    /
#
# Таким образом, количество свободных блоков по 1024 байта
# будет находиться в четвертом поле слева.
# Подобный формат вывода принят в системах linux и bsd
my(@dir,$free);
@dir=`df -k .`;
$free=(split(/\s+/, $dir[$#dir]))[3];
§free*=1024;
```

Обратите внимание на отличие этого фрагмента кода от предыдущего. Для определения объема свободного пространства жесткого диска в системе Windows использовалась команда dir, а в UNIX — команда df -к .. Последняя строка, выводимая командой df -к ., разбивается на части, после чего извлекается содержимое четвертого (по счету) поля и присваивается переменной $free. Однако следует учитывать тот факт, что в различных системах UNIX выводимые командой df данные имеют разный формат (отличается количество полей) или поля расположены в другом порядке. Поэтому, если вы столкнетесь с такой проблемой, внесите необходимые коррективы в программу на Perl.

Итак, мы создали две программы для двух операционных систем Windows и UNIX, выполняющих одни и те же действия, — определение объема свободного пространства на диске. Теперь давайте объединим их в одну универсальную программу, которая будет работать и в Windows, и в UNIX.

```
if ($^O eq 'MSWin32') {
   # В последней строке сообщения команды 'dir' должно находиться нечто
   # наподобие: 10 dir(s)   67,502,080 bytes free
   # В системе Windows 98 вместо слова 'bytes' может выводиться 'MB'
   my(@dir,$free);
   @dir=`dir`;
   $free=$dir[$#dir];
   $free=~s/.*([\d,]+) \w+ free/$1/;
   $free=~s/,//g;
} elsif ($^O eq 'linux') {
   # В последних строках сообщения команды 'df -k'
   # должно находиться нечто наподобие:
   #
   # Filesystem  1K-blocks    Used    Avail Capacity  Mounted on
   # /dev/wd0s1a     31775    21431     7802    73%     /
   #
   # Таким образом, количество свободных блоков по 1024 байта
   # будет находиться в четвертом поле слева.
   # Подобный формат вывода принят в системах linux и bsd
   my(@dir,$free);
   @dir=`df -k .`;
   $free=(split(/\s+/, $dir[$#dir]))[3];
   $free*=1024;
} else {
  warn "Неизвестный тип операционной системы\n";
}
```

Теперь наша универсальная программа включает версии для DOS/Windows и Linux. Если кто-то попытается запустить ее в другой операционной системе, будет выведено соответствующее предупреждение и программа корректно завершит свою работу. Теперь практически вся работа по написанию переносимой программы завершена. Осталось только оформить ее в виде подпрограммы. Тогда все ее внутренние переменные будут локальными, и при необходимости любой программист сможет легко воспользоваться созданной вами подпрограммой.

```
# Подпрограмма вычисления свободного места в текущем каталоге
sub freespace {
 my(@dir,$free);
  if ($^O eq 'MSWin32') {
     # В последней строке сообщения команды 'dir' должно находиться нечто
     # наподобие: 10 dir(s)   67,502,080 bytes free
     # В системе Windows 98 вместо слова 'bytes' может выводиться 'MB'
     @dir=`dir`;
     $free=$dir[$#dir];
     $free=~s/.*([\d,]+) \w+ free/$1/;
     $free=~s/,//g;
   } elsif ($^O eq 'linux') {
     # В последних строках сообщения команды 'df -k'
     # должно находиться нечто наподобие:
     #
     # Filesystem  1K-blocks    Used    Avail Capacity  Mounted on
     # /dev/wd0s1a     31775    21431     7802    73%     /
     #
     # Таким образом, количество свободных блоков по 1024 байта
```

```
    # будет находиться в четвертом поле слева.
    I Подобный формат вывода принят в системах linux и bsd
    @dir=`df -к .`;
    $free=(split(/\s+/, $dir[$#dir]))[3];
    $free*=1024;
  } else {
    $free=0; # Значение по умолчанию
    warn "Неизвестный тип операционной системы\n";
  }
  return $free;
}
```

Теперь, когда вам понадобится определить, сколько места осталось на диске, вызовите функцию **freespace()**, и она вернет нужное значение. Если же эта функция будет вызвана в той операционной системе, которая ею не поддерживается, на экране появится сообщение об ошибке. Исправить подобную ситуацию довольно просто — добавьте еше один блок **elsif** в условный оператор if.

Резюме

На этом занятии вы узнали, как воспользоваться утилитами операционной системы для выполнения поставленной задачи. Для запуска системной утилиты или конвейера в Perl предназначена функция system. Чтобы получить данные, выведенные утилитой на стандартное устройство вывода, следует поместить команду вызова в обратные кавычки (``). В результате появляется возможность присвоить эти данные переменной для дальнейшей обработки средствами языка Perl. С помощью функции open можно открывать не только файлы, содержащие данные, но и файлы с программами. Для записи операторов программы на Perl в файл используется оператор print, а для чтения — угловые скобки (о). И в конце занятия была продемонстрирована методика написания переносимых программ, которые могут без изменения работать на разных компьютерных платформах. При этом не требуется написание отдельных версий программы для каждой поддерживаемой платформы, поскольку в Perl есть средства, позволяющие программе определить тип платформы, на которой она выполняется.

Вопросы и ответы

Как открыть конвейер, чтобы получить данные **от одной программы, обработать их** и **передать другой программе? Почему не работает оператор наподобие ореп(Р,** "| команда |")?

Решение этой простой задачи на деле оказывается гораздо сложнее, чем может показаться на первый взгляд. Причина заключается в том, что попытка чтения и записи в конвейер со стороны одного и того же процесса приводит к *взаимной блокировке* процессов. В самом деле, после открытия конвейера при попытке выполнить чтение с помощью оператора <P> ваша программа будет переведена в состояние ожидания появления данных от программы *команда.* В то же время программа *команда* будет ожидать появления данных от вашей программы. Таким образом, произойдет взаимная блокировка двух программ, выхода из которой нет. Однако если будет включен режим выдачи предупреждений, интерпретатор Perl выведет сообщение Can't do bidirectional pipe (Нельзя создать двунаправленный конвейер).

Если вы столкнулись с подобной проблемой, воспользуйтесь модулем IPC: :0реп2, с помощью которого можно создать двунаправленный конвейер. Работа с модулями будет описана на 14-м занятии, "Использование модулей".

Почему после выполнения оператора $a=system("команда") переменной $a не присваиваются данные, выведенные командой в стандартный выходной поток, как можно было бы предположить?

Вы перепутали функцию system с обратными кавычками. Эта функция не перехватывает выходные данные программы. Чтобы решить проблему, воспользуйтесь оператором $a=*команда;.

Почему при запуске внешних команд в системе UNIX с помощью обратных кавычек (` `) не перехватываются сообщения об ошибках?

Дело в том, что всем программам в системе UNIX (в том числе и Perl) назначается два стандартных выходных потока: STDOUT и STDERR. Поток STDOUT используется для вывода обычных сообщений во время работы программы, а поток STDERR — для вывода сообщений об ошибках. При помещении команды в обратные кавычки или открытии конвейера с помощью функции open выполняется перехват только потока STDOUT. Чтобы решить проблему, необходимо с помощью средств командной оболочки перенаправить поток STDERR в STDOUT, как показано ниже:

```
$a=`команда 2>&1`; # Запуск команды с перехватом сообщений и ошибок
```

За более подробной информацией, относящейся к процессу перехвата ошибок, обратитесь к 8-му разделу списка часто задаваемых вопросов по Perl. Для этого введите команду perldoc perlfaq8.

Семинар

Контрольные вопросы

1. С помощью какой команды можно постранично отобразить выводимую программой информацию?

   а) perl myprog.pl | more.

   б) open(M, "| more") || die; print M "Данные...\n";

   в) open(M, ">more") || die; print M "Данные...\n";

2. Какое значение переменной $foo будет использоваться в операторе: $r=`dir $foo`?

   а) значение переменной $foo системной оболочки;

   б) значение переменной $foo Perl, после чего будет выполнена команда dir.

3. Решение какой из перечисленных ниже задач зависит от типа операционной системы?

   а) определение свободного дискового пространства;

   б) получение списка файлов каталога;

   в) удаление каталога.

Ответы

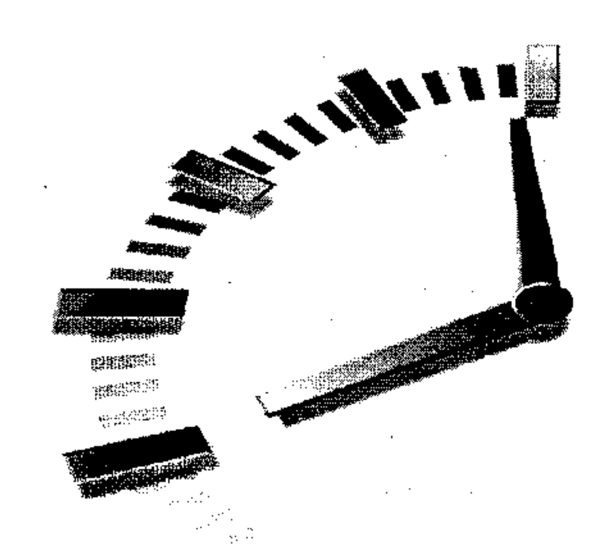
1. Правильными являются варианты а) и б). Если используется вариант а), выходной поток программы myprog.pl перенаправляется на вход программы more. В случае варианта б) данные, записанные в файл с дескриптором `M`, перенаправляются на вход программы more для отображения в постраничном режиме.

2. Правильный ответ — вариант б). Чтобы изолировать переменную $foo от Perl, необходимо воспользоваться оператором qx'dir $foo'.

3. Единственно правильный вариант а), поскольку для получения списка файлов каталога можно воспользоваться операторами Perl glob, <*> или opendir/readdir/closedir, а для удаления каталога служит команда Perl `rmdir`.

Упражнения

- С помощью функций получения статистических данных, описанных на 8-м занятии, "Функции", модифицируйте программу из листинга 11.1 так, чтобы она отображала более подробную информацию о файлах.
- Если вы работаете в системе UNIX, отличной от Linux, добавьте в подпрограмму freespace() соответствующую ветку для определения свободного пространства на диске. В качестве отправной точки воспользуйтесь примером для Linux.



12-й час

Работа с командной строкой Perl

На предыдущих занятиях мы рассматривали Perl как обычный интерпретатор, и не более того. Другими словами, команда perl использовалась исключительно для выполнения программ на Perl, сохраненных в отдельных файлах. Однако этим далеко не исчерпываются все ее возможности. Оказывается, что в интерпретатор встроен также отладчик, который позволяет выполнять программы на Perl подобно воспроизведению видеокассеты, — вы можете "перематывать" программу в начало, выполнять ее в замедленном режиме и останавливать в любом месте для анализа содержимого внутренних переменных. В некоторых случаях отладчик просто незаменим при поиске изощренных ошибок в программах на Perl.

Интерпретатор Perl позволяет также запускать программы, которые не хранятся в файле. Например, вы можете ввести короткую программу прямо в командной строке и выполнить ее.

Основные темы этого занятия.

- Использование отладчика Perl.
- Запуск небольших программ на Perl прямо из командной строки.

Отладчик Perl

В Perl отладчик встроен прямо в интерпретатор. Он позволяет запустить любую программу на Perl в пошаговом режиме, просматривать и изменять значения ее внутренних переменных, прерывать выполнение программы и возобновлять ее с любого места. При этом, с точки зрения программы, процесс отладки происходит совершенно прозрачно, т.е. отладчик никак не влияет на ее внутреннюю структуру: входные данные по-прежнему поступают с клавиатуры, а вывод осуществляется на экран. Просто программа "не знает", работает она под отладчиком или нет, остановлена или выполняется в пошаговом режиме. Отладчик позволяет полностью контролировать поведение программы, не нарушая ее целостности.

Запуск отладчика

Отладчик Perl запускается из командной строки операционной системы. В системах DOS и Windows в качестве приглашения на ввод команды обычно используется C:\>. В UNIX приглашение командной оболочки появляется после регистрации пользователя в системе и обычно имеет вид % или $. На компьютерах Macintosh для запуска отладчика выберите команду Debugger из меню Script. В результате на экране появится диалоговое окно отладчика.

Во всех примерах данного занятия используется программа Employee (см. листинг 9.1), о которой шла речь на 9-м занятии, "Дополнительные функции и операторы". Поэтому имеет смысл сделать закладку на странице 97, чтобы при необходимости быстро найти текст программы. Для отладки программы Employee после приглашения операционной системы введите приведенную ниже команду (для примера мы использовали систему DOS):

```
C:\> perl -d Employee
```

Ключ -d, указанный в командной строке, переводит интерпретатор Perl в режим отладки. После ключа указывается имя файла, содержащего отлаживаемую программу. В результате на экран будет выведено несколько сообщений, в одном из которых содержится номер версии отладчика, как показано ниже:

```
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(Employee:5):      my @employees=(
main::(Employee:6):           'Смит,Боб,123101,9.35,40',
main::(Employee:7):           'Франклин,Алиса,132912,10.15,35',
main::(Employee:8):           'Войоховиц,Тед,198131,6.50,39',
main::(Employee:9):           'Нег,Венди,141512,9.50,40',
main::(Employee:10):          'Клиборн,Стен,131211,11.25,40',
main::(Employee:11):        );
  DB<1>__
```

Кроме номера версии (в нашем случае 1.07) отображается строка с подсказкой о том, как можно вызвать справочную систему. Далее отладчик отображает первый оператор программы, который состоит из семи строк и начинается с фразы my @employees=(, а заканчивается закрывающейся круглой скобкой и точкой с запятой. Кроме содержимого оператора отладчик выводит дополнительную информацию (название процедуры, имя файла и номер строки в файле), помогающую идентифицировать отлаживаемый оператор.

В последней строке отладчик помещает приглашение DB<1> и устанавливает после него курсор. Цифра 1 означает, что отладчик ожидает ввода первой команды. В описанном нами состоянии программа на Perl не запущена и находится в состоянии ожидания. При этом на экране отображается оператор, который *должен* быть выполнен следующим (в нашем случае my @employees=(), а не ТОТ, который *был* выполнен раньше. Команды отладчика вводятся после приглашения.

Основные команды отладчика

Одной из самых важных команд, которую вы наверняка будете часто использовать, является команда help, предназначенная для вызова справочной системы отладчика. Введите символ h, в результате на экране появится список всех команд отладчика и их краткое описание. Если после команды h указать имя требуемой команды, то отладчик выведет описание только этой команды.

Часто бывает, что описание какой-то команды не помешается на экране. При этом первые несколько строк просто "уедут" с экрана. Чтобы вывести справочную информацию постранично, поместите перед командой символ вертикальной черты |. Например, чтобы вывести справку по всем командам отладчика в постраничном режиме, введите команду |h.

Самой ценной возможностью отладчика Perl является его способность запускать программы в пошаговом режиме. Пользуясь этим, давайте продолжим выполнение упражнения, которое мы начали в предыдущем разделе, и перейдем к следующему оператору программы Employee. Итак, после приглашения отладчика введите команду n (next, или следующий):

```
DB<1> n
main::(Employee:24):                my ($L1, $F1)=split(',', $a>;
```

В результате интерпретатор Perl выполнит первый оператор программы Employee (строки с 5 по 11). После этого отладчик распечатает следующий оператор, который должен быть выполнен, — my ($L1, $F1)=split(',', $a); — и выведет приглашение. В результате выполнения первого оператора программы был проинициализирован массив @employees. В нем будет содержаться информация по пяти сотрудникам. Чтобы распечатать значение элементов массива, воспользуйтесь оператором print, как показано ниже:

```
DB<1> print @employees
Смит,Боб,123101,9.35,40Франклин,Алиса,132912,10.15,35Войоховиц,Тед,198131,6.50,3
9Нег,Венди,141512,9.50,40Клиборн,Стен,131211,11.25,40
```

После приглашения отладчика можно вводить не только его команды, но и любой оператор Perl. Обратите внимание, что в приведенном выше примере элементы массива @employees никак не разделяются и смешаны в кучу. Чтобы распечатать каждое значение в отдельной строке, введите после приглашения такой оператор:

```
DB<2> print join("\n", @employees)
Смит,Боб,123101,9.35,40
Франклин,Алиса,132912,10.15,35
Войоховиц,Тед,198131,6.50,39
Нег,Венди,141512,9.50,40
Клиборн,Стен,131211,11.25,40
```

Чтобы продолжить выполнение программы в пошаговом режиме, вводите каждый раз команду n, как показано ниже:

```
DB<3> n
main::(Employee:23):  @employees=sort {
  DB<3> n
main::(Employee:25):                my ($L2, $F2)=split(',', $b);
  DB<3> n
main::(Employee:26):                return ( $Ы cmp $L2 f Сравним фамилии
```

```
main::(Employee:27):                          ||       # Если они идентичны.
..
main::(Employee:28):                      $F1 cmp $F2 # Тогда сравним имена
main::(Employee:29):              );
  DB<3>
main::(Employee:23):  @employees=sort {
```

Нетрудно заметить, что отладчик, пройдя несколько операторов, снова вернулся к строке 23 так, как будто в программе организован цикл. Все дело в том, что в операторе sort задан блок сортировки, который выполняется в пошаговом режиме для каждого элемента массива **@employees**. Поэтому при каждом вводе команды п отладчик будет выполнять цикл до тех пор, пока не будет отсортирован массив **@employees** (а это произойдет довольно быстро).



> Чтобы повторить предыдущую команду, можно просто нажать клавишу <Enter> в строке приглашения.

Точки останова

Выполнение программы под отладчиком в пошаговом режиме не всегда удобно, например когда программа очень большая. Поэтому для ускорения процесса отладки желательно, чтобы программа на **Perl** выполнялась в обычном режиме и останавливалась при достижении определенного оператора. Места, в которых программа приостанавливает свое выполнение под отладчиком, называются *точками останова*.

Перед тем как задать точки останова, нужно определить номера соответствующих операторов программы. Для этого используется команда 1. С ее помощью можно вывести на терминал 10 следующих строк программы. Повторный ввод команды 1 выводит 10 следующих операторов программы и т.д. Чтобы распечатать листинг программы начиная с определенной строки, введите команду 1 ***#строки***, где вместо параметра ***#строки*** укажите номер строки. В операторе 1 можно также указать диапазон строк, например 1 3-10.

В листинге текущий оператор, который должен быть выполнен, отмечается символом ==>, как показано в следующем примере:

```
  DB<3> 1 23-33
23==>    @employees=sort {
24:                  my ($L1, $F1)=split(', ', $a);
25:                  my ($L2, $F2)=split{', ',$b);
26:                  return { $L1 cmp $L2 # Сравним фамилии
27                                ||      # Если они идентичны...
28                            $F1 cmp $F2 * Тогда сравним имена
29                   );
30                     } @employees;
31
32:      foreach(@employees) {
33:                print_emp($_]);
  DB<4>
```

В данном примере подходящим местом для установки точки останова будет строка 33. Дело в том, что она находится сразу после оператора sort и в ней расположен первый оператор блока основного цикла программы. Точки останова можно задать в

любой строке программы, главное, чтобы в них находился корректный оператор Perl. Нельзя устанавливать точки останова на скобках (строка 30), знаках пунктуации (строка 29), пустых строках (строка 31) и комментариях, если они занимают всю строку.

Для задания точек останова используется команда b *точка_останова*. Вместо параметра *точка_останова* следует указать номер строки или имя подпрограммы. Например, чтобы задать точку останова в строке 33, введите следующую команду:

```
DB<4> b 33
DB<5>
```

Еще одной полезной командой, которая может пригодиться при работе с точками останова, является команда c (continue, или продолжить). При получении этой команды отладчик возобновляет выполнение программы до достижения следующей точки останова или конца программы (в зависимости от того, что произойдет раньше), например:

```
DB<5> c
main::(Employee:33):            print_emp($_);
DB<5>
```

Как и следовало ожидать, в данном случае выполнение программы будет остановлено в строке 33 перед вызовом функции print_emp. Для продолжения работы введите команду c. Поскольку точка останова остается все время активной, после выполнения функции print_emp программа будет снова остановлена в строке 33. При этом на экран будут выведены как данные программы, так и сообщение отладчика:

```
DB<2> c
198131 Тед Войохович         6.50   39   253.51
main::(Employee:33):            print_emp($_);
```

Для получения списка точек останова следует воспользоваться командой L, как показано ниже на примере:

```
DB<2> L
Employee:
 33:            print_emp($_);
   break if (1)
```

Из этого примера видно, что в отладчике была установлена одна точка останова в строке 33 файла Employee.

Для удаления точек останова используется команда d, синтаксис которой такой же, как и у команды b — d *#строки* или d *имя_продпрограммы*. Вот пример:

```
DB<2> d 33
DB<3>
```

Другие команды отладчика

Предположим, мы хотим отладить функцию print_emp и заодно посмотреть, как она работает. Это можно сделать несколькими способами. Для начала давайте перезапустим программу с помощью команды R:

```
DB<3> R
Warning: some settings and command-line options may be lost!
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
```

```
Editor support available.

Enter h or `h h' for help, or `perldoc perldebug' for more help.

main::(Employee:5):      my @employees=(
main::(Employee:6):           'Смит,Боб,123101,9.35,40',
main::(Employee:7):           'Франклин,Алиса,132912,10.15,35',
main::(Employee:8):           'Войоховиц,Тед,198131,6.50,39',
main::(Employee:9):           'Нег,Венди,141512,9.50,40',
main::(Employee:10):          'Клиборн,Стен,131211,11.25,40',
main::(Employee:11):     );
  DB<1> b 33
```

Команда R выполняет установку программы в исходное состояние и подготавливает ее к повторному запуску. При этом аннулируются все заданные ранее точки останова, а значения переменных очищаются. Поэтому в предыдущем примере мы восстановили точку останова в строке 33. Теперь запустим программу на выполнение с помощью команды с:

```
  DB<2> c
main::(Employee:33):                print_emp($_);
  DB<2>
```

Перед вызовом функции print_emp программа остановится. А теперь, чтобы выполнить следующий оператор в пошаговом режиме, воспользуемся командой п:

```
  DB<2> п
198131 Тед Войоховиц         6.50   39   253.51
main::(Employee:32):    foreach(@employees) {
  DB<2> n
main::(Employee:33):                print_emp($_);
  DB<2>
```

Что произошло? Почему мы не увидели, как выполняются в пошаговом режиме операторы внутри функции print_emp? Дело в том, что команда п выполняет текущий оператор, не выполняя трассировку функции. Для того чтобы "зайти" внутрь функции, используйте команду s (step, или шаг вперед). Данная команда аналогична команде п, но, в отличие от нее, выполняет оператор вызова функции и останавливается на *ее* первом операторе, а не выполняет функцию до конца и останавливается на следующем операторе *после* вызова функции. Вот пример:

```
main::(Employee:33):                print_emp($_);
  DB<2> s
main::print_emp(Employee:14):       my ($last,$first,$emp,$hourly,$time)=
main::print_emp(Employee:15):                   split(',',$_[0]);
  DB<2>
```

Как видите, отладчик остановил программу перед выполнением первого оператора функции print_emp. Заметьте, что того же эффекта можно было бы добиться, установив точку останова на функцию print_emp с помощью команды b print_emp. Для того чтобы продолжить выполнение операторов функции в пошаговом режиме, воспользуйтесь командой п, как показано ниже:

```
  DB<2> n
main::print_emp(Employee:16):       my $fullname;
  DB<2> n
main::print_emp(Employee:17):        $fullname=sprintf("%s %s", $first, $last);
```

```
  DB<2> n
main::print_emp(Employee:18):      printf("%6d %-20s %6.2f %3d %7.2f\n",
main::print_emp(Employee:19):             $emp, $fullname, $hourly, $time,
main::print_emp(Employee:20):              ($hourly * $time)+.005);
  DB<2>
```

Во время выполнения программы в отладочном режиме можно на ходу изменять значения переменных. Например, чтобы временно увеличить сотруднику почасовую тарифную ставку на $2,5, используются следующие команды:

```
  DB<2> print $hourly
11.25
  DB<3> $hourly+=2.5

  DB<4> print $hourly
13.75
  DB<5> n
131211 Стен Клиборн           13.75   40   550.01
main::(Employee:32):  foreach(@employees) {
  DB<5>
```

В этом примере сначала мы распечатали значение переменной `$hourly` (11.25), затем увеличили его на 2.5 и продолжили выполнение программы. Обратите внимание, что оператор printf распечатал уже новое значение переменной `$hourly`.

И, наконец, чтобы завершить работу отладчика, введите команду q.

Упражнение: поиск ошибки

В этом упражнении мы рассмотрим методику поиска ошибок в программе с помощью отладчика. В программе, приведенной в листинге 12.1, есть ошибка (точнее, целых две). Предполагается, что программа должна выводить в цикле приведенные ниже сообщения:

```
В корзине осталось 20 пачек мороженого
В корзине осталось 19 пачек мороженого
...
В корзине осталось 1 пачка мороженого
В корзине осталось 0 пачек мороженого
```

Но она почему-то отказывается это делать. Итак, ваша задача — набрать программу, текст которой приведен в листинге 12.1, и попытаться найти в ней ошибки. Обратите внимание, что ни одна из ошибок не относится к категории синтаксических, поскольку интерпретатор Perl не выводит никаких предупредительных сообщений (а ведь используется ключ `-w` и оператор use strict). В то же время с помощью отладчика поиск ошибок не составит особого труда.

Итак, после набора текста программы запустите ее в отладочном режиме. Не забывайте время от времени распечатывать значения ключевых переменных и выражений, а также обходить трассировку вызовов функции message.

Листинг 12.1. Программа с ошибками

```
#!/usr/bin/perl -w
# В этой программе содержится ДВЕ ошибки. Найдите их
use strict;

sub message {
```

```
    my($quant)=@_;
    my($mess,$repl);
    $repl="пачек";
   $mess="В корзине осталось $quant пачек мороженого\n";

  if (($quant < 5) and ($quant >1 )) {
        $repl="пачки";
  }
  if ($quant eg 1) {
        $repl="пачка";
  }
   $mess=s/пачек/$repl/;
  print $mess;
}

foreach(20..0) {
  &message($_);
}
```

Ответ вы найдете ниже, в разделе "Семинар".

Дополнительные возможности интерпретатора

Отладчик это не единственная возможность интерпретатора Perl, которую можно активизировать из командной строки. В действительности на Perl можно написать множество полезных программ, поместив их прямо в командую строку вызова интерпретатора.



Пользователи Macintosh должны выполнять приведенные ниже примеры упражнений с командной строкой, выбрав в меню Script пункт **1-liners**. После этого можно вводить текст команды в появившееся диалоговое окно.

Однострочные программы

Чтобы выполнить простенькие операторы на Perl прямо из командной строки, необходимо поместить их после ключа -е. В командной строке вы можете указать любой допустимый оператор **Perl**, как показано в следующем примере:

```
C:\>perl -e "print 'Hello, world';"
Hello, world
```

Чтобы ввести в командной строке несколько операторов Perl, можно использовать несколько ключей -е или разделить операторы точкой с запятой. Вот пример:

```
C:\>perl -e "print 'Hello, world';" -e "print 'Goodbye, world';"
Hello, worldGoodbye, world
```

Не забывайте, что большинство командных интерпретаторов накладывают определенные ограничения на использование кавычек и служебных символов. Например, в интерпретаторах Windows/DOS и Windows NT — command.com и cmd.exe — разрешается

использовать двойные кавычки для группировки слов, как в рассмотренных выше примерах. Однако если нужно поместить в двойные кавычки служебные символы, такие как <,>, | или ^, а также слова в двойных кавычках, оказывается, что сделать это не так-то просто. За дополнительной информацией по этой теме обратитесь к справочному руководству по конкретному командному интерпретатору.

В системе UNIX необходимо соблюдать правило — каждой открывающей кавычке должна соответствовать закрывающая кавычка. Другими словами, в UNIX использование кавычек должно быть сбалансированным. Если же необходимо поместить один из служебных символов внутрь кавычек, перед ним необходимо поставить обратную косую черту \, например:

```
$ perl -e 'print "Hello, World\n";' -e 'print "Goodbye, world\n";'
```

Эта команда должна работать в большинстве оболочек UNIX, таких как sh, csh, ksh, bash и др. При этом сообщения должны выводиться с новой строки. Чтобы получить подробную информацию о правилах использования служебных символов в командных строках, обратитесь к справочной странице соответствующей оболочки UNIX.

Одной из полезных и часто используемых возможностей является комбинирование ключей -e и -d в командной строке интерпретатора Perl. Это позволяет перевести интерпретатор в режим отладки без необходимости загрузки программы, например:

```
C:\> perl -d -e 1
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or `h h' for help, or `perldoc perldebug' for more help.

main::(-e:1):   1
  DB<1>
```

После ввода этой команды отладчик отображает приглашение и переходит в режим ожидания команд от пользователя. Обычно этот режим работы используется для тестирования операторов Perl на предмет правильности синтаксиса, когда не требуется писать отдельную программу. Просто наберите нужный оператор Perl после приглашения отладчика, нажмите клавишу <Enter>, и он будет выполнен. Введенная в командной строке единица (1) является на самом деле простейшей программой на Perl. Это обычное выражение, значение которого равно 1.

Дополнительные ключи командной строки

КЛЮЧ -С позволяет провести синтаксический анализ программы без ее запуска на выполнение, например:

```
C:\> perl -c Employee
Employee syntax OK
```

Если в программе содержится синтаксическая ошибка, сообщение будет другим, как показано ниже:

```
C:\> perl -c Employee
syntax error at Employee line 13, near ")
```

```
sub print_emp "
Can't use global @_ in "my" at Employee line 15, near ",$_"
syntax error at Employee line 21, near "}"
Employee had compilation errors.
```

При комбинировании ключей -w и -c производится компиляция программы с включенным режимом вывода предупреждений.

Очень часто в разговоре опытных программистов на Perl проскакивают слова о номере версии интерпретатора. Номер версии может также спросить у вас системный администратор при помещении созданных вами программ на Web-сервер. Чаще всего, конечно, используется версия Perl 5. Номер версии интерпретатора можно определить с помощью ключа -v, как показано ниже на примере:

```
C:\> perl -v
This is perl, v5.6.0 built for MSWin32-x86-multi-thread
(with 1 registered patch, see perl -V for more detail)

Copyright 1987-2000, Larry Hall

Binary build 613 provided by ActiveState Tool Corp. http://www.ActiveState.com
Built 12:36:25 Mar 24 2000


Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5.0 source kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using `man perl' or `perldoc perl'. If you have access to the
Internet, point your browser at http://www.perl.com/, the Perl Home Page.
```

В нашем примере мы использовали версию интерпретатора 5.6.0 для Windows/DOS. Чтобы получить подробную информацию по конкретной версии интерпретатора (как и когда она была скомпилирована, каковы были параметры компиляции и т.д.), запустите Perl с ключом -V, например:

```
C:\> perl -V
Summary of my perl5 (revision 5 version 6 subversion 0) configuration:
  Platform:
    osname=MSWin32, osvers=4.0, archname=MSWin32-x86-multi-thread
...
  Compiler:
    cc='cl', optimize='-O1 -MD -DNDEBUG', gccversion=
...
Characteristics of this binary (from libperl):
  Compile-time Options: MULTIPLICITY USE_ITHREADS PERL_IMPLICIT_CONTEXT PERL_IMP
LICIT_SYS
  Locally applied patches:
        ActivePerl Build 613
  Built under MSWin32
  Compiled at Mar 24 2000 12:36:25
  @INC:
    E:/Tools/Perl/lib
    E:/Tools/Perl/site/lib
    .
```

Эта информация может пригодиться при выяснении проблем, возникших с конкретным интерпретатором Perl. Возможно, с ее помощью вам удастся определить, что ваш интерпретатор был некорректно установлен. В конце листинга обратите внимание на строку @INC. В ней перечислены полные пути к каталогам, в которых интерпретатор Perl будет искать свои модули. Таким образом, после установки интерпретатор Perl нельзя просто взять и переместить в другой каталог, поскольку при этом он "потеряет" все свои модули. О том, что такое модули, мы поговорим на 14-м занятии, "Использование модулей".

Угловой оператор и однострочные программы

На одном из предыдущих занятий мы рассматривали угловой оператор и говорили о том, что он используется для двух целей.

1. С помощью оператора <> осуществляется ввод данных из файла, дескриптор которого помещен в угловые скобки, например <STDIN>.
2. Если поместить в угловые скобки шаблон, например <*.bat>, то в результате будет возвращен список файлов, соответствующих этому шаблону. Напомним, что такая операция называется *отбором файлов*.

Однако у рассматриваемого нами оператора есть еще одна полезная функция. Если в угловом операторе не указать дескриптор файла (о), то данный оператор будет читать содержимое всех файлов, указанных в командной строке. Если же в командной строке файлы не указаны, то информация будет считываться из стандартного входного потока. Иногда за свою форму угловой оператор без дескриптора файла (о) программисты называют "бубновым оператором". А теперь давайте в качестве примера рассмотрим следующую простую программу:

```
#!/usr/bin/perl -w

while(<>) <
   print $_;
>
```

Сохраните программу в файле Exaraple.pl и запустите ее на выполнение с помощью команды

```
C:\> perl -w Example.pl filel file2 file3
```

В результате угловой оператор будет построчно считывать содержимое сначала filel, затем file2 и наконец file3. Если вы опустите имена файлов, то считывание информации будет происходить из стандартного входного потока. Подобное поведение программы полностью соответствует поведению утилит системы UNIX, таких как sed, awk и др.



> Аргументы, указанные в командной строке при вызове интерпретатора Perl (те, что расположены после ключей -w, -c, -d и -e), автоматически проходят стадию синтаксического анализа и помещаются в массив @ARGV. Например, в предыдущем примере элемент $ARGV[0] будет равен "filel", $ARGV[1] — "file2" и т.д.

Ключ -n позволяет поместить оператор, указанный после ключа -е, в следующую небольшую программу:

```
LINE:
while(<>) {
  ... # Оператор, указанный после ключа -e
}
```

Таким образом, чтобы создать однострочную программу, удаляющую из введенных строк начальные пробелы, можно воспользоваться следующей командой:

```
C:\> perl -n -e 's/^\s+//g; print $_;' filel
```

В результате Perl выполнит такой фрагмент программы:

```
LINE:
while(<>) {
   s/^\s+//g;
   print $_
}
```

В нашем примере файл filel открывается и его содержимое в цикле while считывается построчно в переменную $_. Затем прочитанная строка редактируется с помощью шаблона s/^\s+//g и выводится на печать.

Вместо ключа -п можно использовать ключ -р. Тогда после редактирования с помощью шаблона строка будет автоматически выводиться на печать. Следовательно, предыдущую команду можно переписать так:

```
C:\> perl -p -e 's/^\s+//g;' filel
```

При редактировании файлов с помощью однострочных программ нельзя одновременно открывать один и тот же файл как для чтения, так и для записи, как показано в следующем примере:

```
C:\> perl -p -e 's/^\s+//g;' dosfile > dosfile
```

В этом примере производится попытка удалить из строк файла dosfile начальные пробелы. Однако проблема заключается в том, что содержимое файла dosfile будет затерто еще до того, как программа на Perl начнет выполняться. Чтобы редактирование файла выполнялось правильно, необходимо перенаправить выходной поток в другой файл, а затем присвоить ему первоначальное имя, как показано в следующем примере:

```
C:\> perl -p -e 's/^\s+//g;' dosfile > tempfile
C:\> rename tempfile dosfile
```



Для некоторых энтузиастов Perl написание коротких однострочных программ является приятным хобби. Идея состоит в том, что чем короче и полезнее программа, тем лучше. Массу примеров подобных однострочных программ можно найти на страницах выходящего раз в квартал *The Peri Journal.*

Резюме

На этом занятии речь шла о том, как с помощью отладчика можно быстро и эффективно отладить программу на Perl. Кроме того, была рассмотрена дополнительная возможность углового оператора, с помощью которого можно обработать содержимое всех файлов, указанных в командной строке Perl. В заключение говорилось о том, как с помощью ключей -п и -р выполнять короткие, но полезные однострочные программы на Perl.

Вопросы и ответы

Я привык пользоваться отладчиком с графическим интерфейсом. Существует ли подобный отладчик для программ на Perl?

Да, причем несколько. В поставку версии Perl для Windows входит совсем неплохой графический отладчик.

Что обозначает префикс `main::`, который выводит отладчик перед именем программы?

Отладчик Perl выводит имена в соответствии с принятой концепцией присвоения имен. Некоторые из ее элементов мы будем рассматривать на следующих занятиях, поэтому пока не обращайте на них внимания.

Есть ли у интерпретатора Perl какие-либо другие ключи, которые не были рассмотренные на этом занятии?

Есть, причем несколько. Полный список ключей приведен в электронном справочном руководстве. Чтобы ознакомиться с ним, введите команду `perldoc perlrun`.

Семинар

Контрольные вопросы

1. Найдите ошибки в программе, приведенной в листинге 12.1.
2. Если в командной строке не указаны имена файлов, оператор <> возвращает:

   а) значение `undef`;

   б) строки, поступающие из стандартного входного потока;

   в) значение true.
3. Отладчик Perl может распечатывать операторы программы перед их выполнением. Такой режим его работы называется трассировкой. Как перевести отладчик в режим трассировки? *(Подсказка:* для ответа на этот вопрос посмотрите сообщение, которое выводит отладчик в ответ на команду h.):

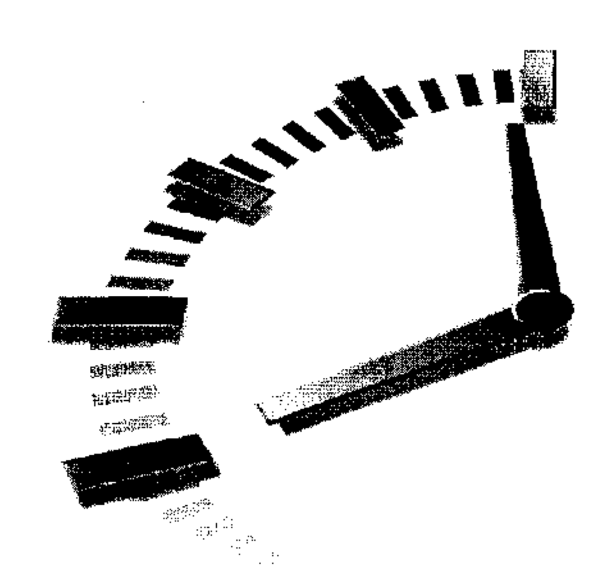
   а) команда T;

   б) команда t.

Ответы

1. Первая ошибка в строке 21. Там ошибочно указан диапазон `(20..0)`. В операторе диапазона .. не допускается использование обратного счета. Поэтому оператор цикла в строке 21 нужно заменить на такой: `for($_=20; $_>-1; $_--)`. Можно также инвертировать список (0..20) с помощью оператора reverse.

   Вторая ошибка в строке 17 в операторе `$mess=s/пачек/$repl/;`. При программировании подразумевалось, что необходимо выполнить замену текста в строке, которая содержится в переменной $mess. Но вместо этого был указан оператор присваивания переменной `$mess` результата подстановки текста в переменной $_. Чтобы исправить ошибку, замените оператор присваивания = на оператор подстановки =".
2. Правильный ответ — вариант б). Если в командной строке не указаны имена файлов, оператор <> выполняет чтение строк текста из стандартного входного потока `STDIN`.
3. Правильный ответ — вариант б). Режим трассировки активизируется с помощью команды t. Команда T предназначена для выполнения *трассировки стека.* Трассировка стека позволяет определить последовательность вызовов подпрограмм, т.е. в каком порядке одна подпрограмма вызывала другую.



13-й час

Структуры и ссылки

Материал этого занятия может быть интересен тем, кто только начал изучать программирование и для кого Perl — первый язык программирования. В большинстве языков предусмотрены средства, с помощью которых можно из одной области памяти сослаться на данные, находящиеся в другой области памяти. В одних языках программирования (Pascal и С) эти средства называются *указателями (pointers),* в других (ассемблер) — *косвенными ссылками (indirect references),* а в BASIC или Java, например, подобные средства вообще не предусмотрены. Поэтому, если раньше вы никогда не пользовались ссылками, указателями или косвенными ссылками, внимательно несколько раз прочтите вступительный раздел данного занятия, поскольку он очень труден для восприятия.

В Perl также предусмотрены подобные средства, которые называются *ссылками (references).* Ссылки в Perl используются для разных целей, однако на данном занятии мы сосредоточим свое внимание на вызовах функций, которым передается несколько аргументов сложных типов, а также на том, как можно создать сложный тип данных, например список списков.

Ссылку можно сравнить со старым библиотечным каталогом. Каждой бумажной карточке в каталоге соответствует книга на одной из полок библиотеки. В карточке указывается, к какой категории относится книга (художественная литература, техническая, справочник) и где она расположена в библиотеке. В некоторых библиотечных каталогах можно найти несколько ссылок на одну и ту же книгу, которая относится к разным категориям, а иногда можно даже найти ссылку из одной карточки на другую (типа см. также...).

Ссылки в Perl напоминают бумажные карточки библиотечного каталога, т.е. ссылки указывают на отдельные участки памяти, содержащие данные. С помощью ссылки можно точно определить тип данных (скаляр, массив или хэш), на который она указывает, и место их расположения. Ссылки можно свободно копировать, причем эта операция никак не влияет на данные, на которые они указывают. Можно даже сделать так, чтобы на один и тот же участок памяти указывало несколько ссылок, или создать ссылку на ссылку.

А теперь, вооружившись теорией, внимательно прочтите несколько следующих страниц. Итак, пока у вас ясная голова, на этом занятии мы рассмотрим следующие темы.

- Основные сведения о ссылках.
- Создание структур данных из ссылок.
- Короткий пример, который поможет понять материал.

Основные сведения

Обычные скалярные переменные создаются с помощью оператора присваивания, например:

```
$a="Скаляр"; # Это обычный скаляр
```

После выполнения этого оператора Perl создаст скалярную переменную $a и присвоит ей строковое значение "Скаляр". Итак, до сих пор вы не встретили ничего необычного, не правда ли? Можете считать, что в компьютере выделяется участок памяти, называемый $a, который содержит строковое значение, как показано ниже.

```
  $a
[Скаляр]
```

Теперь, если вы присвоите значение скаляра $a другому скаляру, например $Ь с помощью оператора `$b=$a`, в памяти компьютера создается две одинаковые копии данных, которым присваиваются разные имена, как показано ниже.

```
  $a        $b
[Скаляр]  [Скаляр]
```

Подобный оператор присваивания может пригодиться в том случае, если вы действительно хотите иметь две независимые копии данных. Однако чаще всего нужно, чтобы обе переменные $a и $Ь соответствовали одному и тому же участку данных, а не его копии. В этом случае **следует** создать *ссылку*. Ссылка является просто указателем на некоторый участок памяти. Не забывайте, что она *не содержит* никаких реальных данных! Ссылка обычно хранится в скалярной переменной.

Чтобы создать ссылку на скалярную переменную, перед именем переменной следует поместить символ обратной косой черты. Например, чтобы создать ссылку $ref на переменную $a, используется такой оператор присваивания:

```
$ref=\$a; # Ссылка на переменную $a
```

В результате в памяти компьютера создается такая структура:

```
$ref      $a
[ o ]-->[Скаляр]
```

При этом в переменной $ref не содержится никаких реальных данных; в эту переменную помешается указатель (или адрес) переменной $a. Обратите внимание, что при создании ссылки значение переменной $a не изменяется. После создания ссылки с переменной $a можно выполнять те же действия, что и с обычной переменной, например присваивать значения (`$a="строка"`) или распечатывать (`print $a`).

Поскольку в переменной $ref содержится ссылка на переменную $a, а не на сами данные, с ней нельзя выполнять те же действия, что и с обычными переменными. Например, если вы попытаетесь распечатать значение переменной $ref с помощью

оператора print, то получите нечто вроде SCALAR(0x0000). Чтобы получить доступ к содержимому переменой $a через ссылку $ref, необходимо выполнить так называемую операцию *разыменования* переменной $ref. Представьте себе, что на предыдущем рисунке вы должны двигаться по стрелке к переменной $a. Это и будет разыменование. Например, чтобы распечатать значение переменной $a через ссылку $ref, к ссылке необходимо добавить дополнительный символ $:

```
print $$ref;
```

В этом примере, естественно, подразумевается, что переменная $ref является ссылкой, а дополнительный знак доллара говорит интерпретатору о том, что это ссылка на переменную скалярного типа. В результате выбирается и распечатывается значение переменной, на которое указывает переменная $ref.

С помощью ссылки можно также изменить первоначальное значение переменной. Это еще одно преимущество использования ссылок по сравнению с переменными-копиями. В следующем примере значение переменной $a будет изменено:

```
$$ref="Новое значение";
```

Данный оператор будет выполняться по следующей схеме:

$ref $a

Новое значение

Если по ошибке в операторе присваивания вы укажете $ref вместо $$ref:

```
$ref="Ошибка!";
```

ссылка на переменную $a аннулируется и переменной $ref присваивается реальное значение — строка "Ошибка!", как показано ниже.

$ref $a

Ошибка! Скаляр

После выполнения этого оператора переменная $ref становится обычным скаляром, содержащим текстовое значение. Ссылку можно присвоить другой переменной с помощью оператора присваивания так же, как это происходит со скалярами, например:

```
$name="Евгений";
$nref=\$name;    # Ссылка на переменную $name
$oref=$nref;     # Еще одна ссылка на переменную $name
```

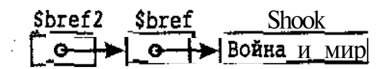
В результате получается следующая схема:



В этом примере для получения значения переменной $name ("Евгений") можно использовать две ссылки $$nref и $$oref. Кроме того, можно создать ссылку на ссылку, как показано ниже на примере:

```
$book="Война и мир";
$bref=\$book;    # Ссылка на переменную $book
$bref2=\$bref;   # Ссылка на переменную $bref (не на $book!)
```

В данном случае цепочка ссылок будет выглядеть так:



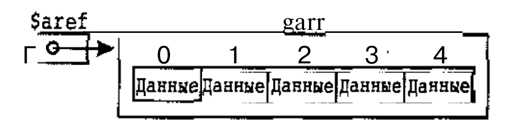
Теперь, для того чтобы добраться до значения переменной $book через ссылку $bref2, нужно использовать конструкцию $$$bref2. Обратите внимание, что для ссылки на переменную Sbook через $bref использовалась конструкция $$bref. Таким образом, "лишний" знак доллара обозначает, что для доступа к оригинальному значению переменной Sbook нужно сделать дополнительную операцию разыменования.

ССЫЛКИ на массивы

Ссылки могут быть также созданы на массивы и хэши. Причем делается это так же, как и ссылка на обычную переменную, — перед именем массива или хэша в операторе присваивания нужно поставить обратную косую черту, например:

```
$aref=\@arr;
```

В результате скалярная переменная $aref будет содержать ссылку на массив @arr. Визуально это можно представить в виде следующей диаграммы:



Для доступа к элементам массива @arr через переменную-ссылку $aref можно использовать один из приведенных ниже операторов:

$$aref[0]	Первый элемент массива @arr
@$aref[2,3]	Сечение массива @arr
@$aref	Весь массив @arr

Для улучшения читабельности программы можно поместить переменную, ссылающуюся на массив, в фигурные скобки, как показано ниже на примерах.

S$aref[0]	тоже, что и	${$aref}{0]
@$aref[2,3]	то же, что и	@{$aref}[2,3]
@$aref	тоже, что и	@{$aref}

Например, в следующем фрагменте кода распечатываются все элементы массива @arr с помощью переменной-ссылки @aref.

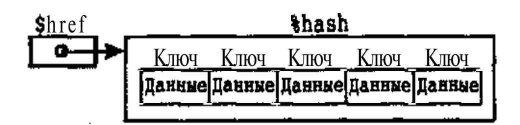
```
foreach {element (@{$aref}) {
    print $element;
}
```

ССЫЛКИ на хэши

Ссылка на хэш создается точно так же, как и на обычный скаляр или массив: нужно поместить перед именем хэша обратную косую черту:

```
$href=\%hash;
```

В результате скалярная переменная $href будет содержать ссылку на хэш @hash. Визуально это можно представить в виде следующей диаграммы:



Для доступа к элементам хэша **%hash** через переменную-ссылку **$href** можно использовать один из приведенных ниже операторов.

$$href{key}	Отдельный элемент хэша **%hash**, адресуемый ключом key. Допустима также конструкция **${href}**{key}
%$href	Весь хэш **%hash**. Допустима также конструкция **%${href}**

Например, для распечатки всех элементов хэша **%hash** с помощью переменной-ссылки **@href** можно воспользоваться следующим фрагментом кода:

```
foreach $key (keys %$href) {
    print $$href{$key}; # То же, что в $hash{$key}
}
```

ССЫЛКИ на аргументы

После того как мы рассмотрели ссылки на массивы и *хэши*, пришло время поговорить о том, как можно передать в подпрограмму несколько массивов или хэшей.

Из материала 8-го занятия, "Функции", вы уже знаете, что приведенный ниже фрагмент кода не работает.

```
# Внимание! Данный код не работает 1
sub getarrays {
    my («a, @b)=@_;
 ...
}
@fruit=qw{яблоко апельсин банан};
fcveggies=qw{морковь кабачок редис};
getarrays(@fruit, @veggies);
```

Этот фрагмент кода не работает потому, что в операторе вызова подпрограммы **getarrays(@fruit,** Gveggies) второй массив **@veggies** затеняется первым массивом **@fruit**, и элементы обоих массивов присваиваются одному массиву **@_**. Внутри подпрограммы getarrays выполняется оператор присваивания **(@a, @b)=@_**. В результате все элементы массива **@veggies** окажутся в массиве 8a, а массив **@b** окажется пустым.

Поскольку элементы обоих массивов при передаче в подпрофамму getarrays оказались в массиве **@_**, нельзя определить, где заканчивается один массив и начинается другой. В результате получился один большой массив.

Чтобы разрешить проблему, необходимо воспользоваться ссылками, т.е. в подпрофамму getarrays нужно передавать не два массива, а только ссылки на них. Вот пример:

```
I Данный код прекрасно работает!
sub getarrays {
    my ($fruit_ref, $veg_ref)=@_;
 ...
}
@fruit=qw{яблоко апельсин банан};
@veggies=qw{морковь кабачок редис};
getarrays(\@fruit, \@veggies);
```

Теперь подпрограмме getarrays всегда передаются два параметра, являющиеся ссылками на массивы. При этом размерность этих массивов не имеет никакого значения. Для доступа к элементам массивов внутри подпрограммы getarrays используются переменные-ссылки **$fruit_ref**, **$veg_ref**, как показано ниже.

```
sub getarrays {
    my ($fruit_ref, $veg_ref)=@_;

    print "Фрукты:", join(',', @$fruit_ref);
    print "Овощи:", join(',', @$veg_ref);
}
```

Ниже мы рассмотрим особенности передачи ссылок на скаляры, массивы и хэши в виде параметров подпрограмм. Не забывайте, что при передаче ссылок в подпрограммы у вас появляется возможность модифицировать данные, на которые указывают эти ссылки. Ниже приведены два примера.

```
t Передача значений
sub changehash {
  my(%local_hash)=@_;
  $local_hash{beast}='медведь';
  return;
}

%hash=(fish  => 'акула',
       bird => 'дрозд');
changehash(%hash);
```

```
# Передача ссылок
sub changehash {
  my($href)=@_;

  $$href{beast}='медведь';
return;
>

%hash=(fish  => 'акула',
       bird => 'дрозд');
changehash(\%hash);
```

В примере слева хэш передается обычным образом. При этом массиву **@_** присваиваются пары ключ-значение передаваемого в подпрограмму хэша **%hash**. Внутри подпрограммы changehash элементы массива **@_** копируются в новый хэш **%local_hash**, который модифицируется, после чего подпрограмма завершает свое выполнение. В результате хэш **%local_hash** уничтожается, а первоначальный хэш **%hash** в главной программе остается без изменений.

В примере справа в подпрофамму changehash через массив **@_** передается ссылка на хэш fchash. В начале подпрофаммы ссылка копируется в локальную **переменную** $href, но, несмотря на это, она по-прежнему указывает на хэш **%hash**. Далее внутри подпрофаммы происходит изменение хэша, на который указывает ссылка $href, и подпрофамма завершает свое выполнение. В результате в хэше **%hash** появится новый ключ beast, которому присвоено значение 'медведь'.



При передаче аргументов в подпрофаммы массив §_ на самом деле является *массивом ссылок*. Таким образом, при изменении элементов массива **@_** будут изменяться и значения аргументов, передаваемых в подпрофамму. Однако учтите, что изменение аргументов, переданных в подпрофамму, считается плохим стилем программирования. Поэтому, если нужно модифицировать в подлрофамме передаваемые ей значения, используйте ссылки. В результате при анализе программы сразу становится понятно, что она модифицирует свои аргументы.

Создание структур

Ссылки можно использовать не только для передачи массивов и хэшей в подпрофаммы. В этом разделе речь пойдет о создании с помощью ссылок сложных структур данных. Обратите внимание, что при выполнении операций со ссылками (после того

как они созданы) не требуется наличия самих элементов данных (скаляров, массивов или хэшей), на которые они ссылаются. Тем не менее после создания ссылки Perl сохраняет элементы массива или хэша, даже если они по логике вещей должны быть аннулированы (например, при выходе переменной из зоны видимости).

В* приведенном ниже примере хэш %hash создается внутри блока кода, причем он является локальным для этого блока.

```
my $href;
{
   my %hash=(phone => 'Белл', light => 'Эдисон');
   $href=\%hash;
}
print $$href{light}; # Будет напечатано "Эдисон"
```

Внутри блока скаляру $href присваивается ссылка на локальный хэш %hash. После завершения работы блока ссылка на хэш %hash остается в активном состоянии, несмотря на то что переменная %hash была объявлена локальной для блока, вышла из зоны видимости и должна быть аннулирована. Таким образом, ссылки на скаляры, массивы, хэши и структуры будут существовать даже после выхода этих объектов из зоны видимости блока или подпрограммы. В нашем примере с помощью ссылки $href можно модифицировать хэш %hash вне зоны его видимости (т.е. за пределами блока, в котором он был объявлен).

Если вы внимательно посмотрите на предыдущий блок кода, то увидите, что нет особого смысла объявлять локальный хэш и назначать ему имя %hash, поскольку оно используется только один раз. Для подобных случаев в Perl предусмотрено специальное средство создания ссылок на структуры данных без назначения им промежуточных имен (типа %hash). Речь идет о так называемой *анонимной памяти (anonymous storage),* В приведенном ниже примере создается ссылка на анонимный хэш, которая помещается в переменную $ahref.

```
$ahref={ phone => 'Белл', light => 'Эдисон' };
```

Элементы анонимного хэша помещаются в фигурные скобки. Такая конструкция возвращает ссылку на созданный хэш, которую нужно присвоить переменной-ссылке. Для работы с анонимным хэшем используется методика, описанная выше в разделе "Ссылки на хэши".

Анонимный массив создается с помощью квадратных скобок [], как показано ниже на примере.

```
$aaref=[ qw(яблоко апельсин банан клубника) ];
```

Для работы с анонимным массивом используется методика, описанная выше в разделе "Ссылки на массивы".

Данные, на которые указывают переменные-ссылки, уничтожаются, как только соответствующая переменная-ссылка выходит из области видимости, как показано ниже на примере.

```
{
   my $ref;
   {
     $ref=[ qw(овес рожь пшеница ячмень) ];
   }
   print $$ref[0]; # Будет напечатано "овес"
}
print $$ref[0];   # Ошибка! Переменная $ref вышла из зоны видимости
```

Следует отметить, что если в программе используется оператор use strict, то этот пример будет скомпилирован с ошибками. Причина заключается в том, что Perl считает переменную $ref глобальной, поскольку она используется в последнем операторе, и в то же самое время она объявлена локальной переменной блока, что недопустимо. Если же активизировать режим выдачи предупреждений с помощью ключа -w, то, вероятнее всего, Perl выведет сообщение undefined value (неопределенное значение), даже если при этом не используется оператор use strict.

Описанные выше анонимные хэши и массивы можно объединять в структуры данных, которые будут описаны в следующем разделе. Поскольку для хранения ссылок на хэши и массивы используются одиночные скалярные переменные, их без проблем можно поместить в элемент массива или хэша, как показано ниже на примере.

```
$a=[ qw( рок поп классика ) ];
$b=[ qw( фантастика боевик драма ) ];
$c=[ qw( журнал книга газета ) ];

# Хэш, содержащий ссылки на массивы
%media=(music => $a, film => $b, 'print' => $c);
```

Примеры структур данных

В следующих разделах будут приведены примеры организации структур данных с помощью массивов и хэшей, которые чаще всего используются при программировании.

Список списков, или двумерный массив

Список списков в Perl часто используется для организации структуры данных, называемой *двумерным массивом.* Как известно, обычный массив представляет собой линейный список значений, как показано ниже на рисунке.

[0]	[1]	[2]	[3]
Значение1	Значение2	Значение3	Значение4

Двумерный массив напоминает таблицу, содержащую строки и столбцы, в которой каждый элемент адресуется по номеру строки и столбца подобно координатам точки на плоскости. Первым в адресе элемента двумерного массива указывается номер строки (начиная с нуля), а вторым — номер столбца, как показано ниже.

[0] [0] Данные	[0] [1] Данные	[0] [2] Данные
[1] [0] Данные	[1] [1] Данные	[1] [2] Данные
[2] [0] Данные	[2] [1] Данные	[2] [2] Данные

В Perl не предусмотрено такого понятия, как многомерный массив. Зато подобную структуру данных легко создать с помощью массива ссылок на массивы. Ниже приведен пример создания массива массивов, элементы которого состоят из литералов.

```
@list_of_lists=(
    [ qv( Ford BMW Mercedes ) ],
    ( qv{ Toyota Mazda Mitsubishi) ],
    { qv( Peugeot Renault) ],
);
```

Посмотрите внимательно на этот фрагмент кода. В нем создается **обычный** список @list_of_lists, который состоит из ссылок на другие списки. Для доступа к отдельным элементам внутренних массивов (или ячеек двумерного массива) используется следующий фрагмент кода:

```
$list_of_lists[0][1];  # "BMW", или 1-я строка 2-й элемент
$list_of_lists[1][2];  # "Mitsubishi", или 2-я строка 3-Й элемент
```

Методика определения количества элементов во внешнем списке ничем не отличается от той, которую мы применяли для массивов. Напомним, что речь идет о конструкции $# или об использовании имени массива в скалярном контексте, например:

```
$#list_of_lists;        # Номер последнего элемента массива @list_of_lists: 2
scalar(@list_of_lists); # Число строк в массиве @list_of_lists: 3
```

Чтобы определить количество элементов во вложенных массивах, придется прибегнуть к небольшой хитрости. Конструкция $list_of_lists[1] возвращает ссылку на вторую строку "двумерного" массива @list_of_lists. Если распечатать это значение, то вы получите нечто типа ARRAY(00000). Поэтому, чтобы элементы массива @list_of_lists интерпретировались как вложенные массивы, при обращении к ним необходимо использовать знак @, как показано ниже на примере.

```
scalar( @{$list_of_lists[2]} ); # 3-я строка состоит из 2-х элементов
$#{$list_of_lists[1]};          # Номер последнего элемента во второй строке: 2
```

Для перебора всех элементов списка списков можно использовать следующий фрагмент кода:

```
foreach my $outer (@list_of_lists) {
    foreach my Sinner (@{$outer}) {
        print "$inner ";
    }
    print "\n";
}
```

В список списков можно добавлять новые элементы и строки, как показано ниже на примере.

```
# Новая строка во внешнем списке
push(@list_of_lists, [ qw(Lexus Lincoln Chevrolet) ] );

f Новый элемент в первом вложенном списке
push(@{$list_of_lists[0]}, qw(Audi) );
```

Примеры других структур

В предыдущем разделе была рассмотрена методика создания одной из основных структур Perl — двумерного массива — с помощью списка списков. Поскольку на размеры массивов, скаляров и хэшей в Perl не накладывается какого-либо ограничения, комбинируя их, вы можете создавать структуры данных любой сложности. Ниже перечислено несколько примеров таких структур:

- список, элементы которого являются хэшами;
- хэш, элементы которого являются списками;
- хэш, элементы которого являются хэшами;
- хэши, элементы которых являются списком, а элементы списка являются хэшами, и т.д.

Из-за недостатка места на страницах этой книги мы даже не будем пытаться описать все эти структуры. В электронном справочном руководстве по Perl есть специальный раздел, который называется Perl Data Structure Cookbook. В нем очень подробно описаны все перечисленные выше структуры данных и много других. Правда, язык изложения несколько труден для понимания. Для каждой рассматриваемой структуры в электронной документации приводятся следующие сведения:

- методика объявления структуры на примере литералов;
- способы наполнения структуры данными;
- процесс добавления новых элементов;
- способы доступа к отдельным элементам структуры;
- методика перебора всех элементов структуры.

Для отображения раздела Perl Data Structure Cookbook наберите в командной строке `perldoc perldsc`.

Отладка программ, использующих ссылки

При отладке программ новички часто путаются со ссылками и не могут понять, на структуру какого вида ссылается конкретная ссылка. Масла в огонь подливает и не совсем прозрачный синтаксис операторов со ссылками. Тем не менее в Perl предусмотрены средства, с помощью которых вы быстро поймете, что происходит в программе.

Для начала попытайтесь просто распечатать значение ссылки. При этом Perl отобразит тип структуры данных, на которую указывает ссылка. Например, оператор

```
print $mystery_reference;
```

выведет примерно такое сообщение:

```
ARRAY(0x1231920)
```

Оно означает, что в переменной `$mystery_reference` хранится ссылка на массив. Кроме массива, возможна ссылка на скаляр (SCALAR), хэш (HASH) и подпрограмму (CODE). Чтобы распечатать содержимое массива, на который указывает переменная `$mystery_reference`, необходимо интерпретировать эту ссылку как массив. Вот пример:

```
print join(',', @{$mystery_reference});
```

В отладчике Perl также предусмотрены средства, позволяющие легко определить тип ссылки. Для этого нужно распечатать содержимое переменной-ссылки, как это вы делали при отладке обычных программ. В следующем примере исследуется содержимое переменной-ссылки $ref.

```
  DB<2> print $ref
HASH(0x184b50)
```

Очевидно, что переменная $ref указывает на хэш. В отладчике предусмотрена специальная команда x, с помощью которой можно распечатать значение ссылки и элементы той структуры, на которую она указывает:

```
  DB<3> x $ref
O HASH(0x184b50)
   'apple' => 'fruit'
   'carrot' => 'vegetable'
   'pear' => 'fruit'
```

В данном случае ссылка указывает на хэш, в котором находятся три элемента с ключами 'apple', 'carrot' и 'pear'. Отладчик может также распечатывать и более сложные структуры данных, например списки списков, как показано в следующем примере:

```
  DB<1> x $a
O ARRAY(0xb3300)
   0 ARRAY(0x1e8694)    # Первая строка двумерного массива
      0   'Ford'        # Элементы первой строки
      1  'BMW
      2  'Mercedes'
   1 ARRAY(0x1b74cc)    t Вторая строка двумерного массива
      0   'Toyota'      # Элементы второй строки
      1  'Mazda'
      2  'Mitsubishi'
   2 ARRAY(0x1e8478)    # Третья строка двумерного массива
      0   'Peugeot'     t Элементы третьей строки
      1  'Renault'
      2  'Citroen'
```

В этом примере переменная $a указывает на массив (ARRAY(0xb3300)). В свою очередь элементы этого массива ссыпаются на три других массива— ARRAY(0x1e8694), ARRAY(0x1b74cc) и ARRAY(0x1e8478). А каждый из вложенных массивов содержит по три элемента.

В модуле Data::Dumper предусмотрена специальная функция для распечатки содержимого ссылок. Этот модуль интересен тем, что выводимая им информация имеет формат, который понимает Perl. Таким образом, вы можете сохранить информацию в файле, а затем загружать ее по мере необходимости. В результате вы получите возможность сохранять значения переменных на диске. Модуль Data: :Dumper будет описан на 14-м занятии, "Использование модулей".

Упражнение: еще одна игра — лабиринт

После того как вы узнали столько нового и непонятного об этих странных ссылках и структурах данных, вам нужно немного развлечься. В этом разделе мы рассмотрим упражнение, демонстрирующее использование структур и ссылок на примере простой игры.

Игра создана по образу и подобию классических игр — задача состоит в том, чтобы найти выход из лабиринта. Собственно лабиринт будет очень простым — он состоит из комнат с одной или несколькими дверями. Двери могут располагаться во

всех четырех стенах комнаты (в северном, южном, восточном и западном направлениях). Цель игры — найти секретную комнату, к которой ведет только один правильный путь. Остальные пути ведут в тупик.

Итак, наберите в текстовом редакторе программу, приведенную в листинге 13.1, и сохраните ее в файле под именем Maze. Запустите программу и сыграйте с компьютером в игру, как показано в листинге 13.2.

Листинг 13.1. Полный исходный текст игры в лабиринт

```
#!/usr/bin/perl -w
use strict;

my @maze=(
    [ qw( в юзв зв зю ) ],
    [ qw( св свз юз сю ) ],
    [ qw( сю - сю зс ) ],
    [ qw( св з св з ) ],
);
my %direction=( 'с' , [-1,0], 'ю' , [1,0],
                'в' , [0,1],  'з' , [0,-1],);
my %full=('в' , 'восток', 'с' , 'север',
          'з' , 'запад',  'ю' , 'юг');
my($curr_x, $curr_y, $x, $y)=(0,0,3,3);
my $move;

sub disp_location {
    my($cx, $cy)=@_;
    print "Вы можете пойти на ";
    while($maze[$cx][$cy] =~/([сюзв])/g) {
            print "$full{$1} ";
    }
    print "($maze[$cx][$cy])\n";
}
sub move_to {
    my($new, $xref, $yref)=@_;

    $new=substr(lc($new),0,1);
    if ($maze[$$xref][$$yref]!~/$new/) {
        print "Ошибочное направление $new. \n";
        return;
    }
    $$xref += $direction{$new}[0];
    $$yref += $direction{$new}[1];
}

until( $curr_x == $x and $curr_y == $y) {
    disp_location($curr_x, $curr_y);
    print "Куда идеи? ";
    $move=<STDIN>; chomp $move;
    exit if ($move =~ /^q/);
    move_to($move, \$curr_x, \$curr_y);
}

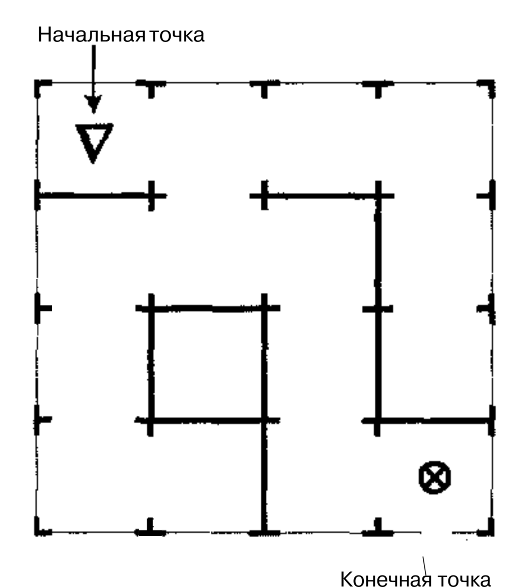
print "Поздравляем! Вы вышли из лабиринта\n";
```

Листинг 13.2. Пример диалога с программой

```
Вы можете пойти на восток (в)
Куда идем? в
Вы можете пойти на юг запад восток (юзв)
Куда идем? ю
Вы можете пойти на север восток запад (свз)
Куда идем? в
Вы можете пойти на юг запад (юз)
Куда идем? ю
Вы можете пойти на север юг (сю)
Куда идем? ю
Вы можете пойти на север восток (св)
Куда идем? в
Поздравляем! Вы вышли из лабиринта
```

 Проведем анализ программы.

- *Строки 1-2.* С этих двух строк начинается практически любая программа на Perl. Ключ `-w` активизирует режим вывода предупреждений, а оператор `use strict` используется для ужесточения контроля интерпретатора Perl над ошибками в программе и выявления плохого стиля программирования.
- *Строки 4—9.* Здесь определяется структура `@maze`, которая описывает лабиринт. Она представляет собой двумерный массив размером 4x4, реализованный в виде списка списков. Каждый элемент этого массива определяет положение дверей в соответствующей комнате лабиринта. Поэтому, если вы собираетесь изменить структуру лабиринта, позаботьтесь о том, чтобы из него был хотя бы один выход. В данном случае структура лабиринта выглядит так:



В одну из комнат (2,1) попасть невозможно, поэтому она отмечена знаком - в структуре `@maze`. На самом деле для этих целей можно использовать любой символ, кроме с, п, з, в.

- *Строки 10-11.* При движении игрока в одном из четырех направлений соответствующим образом должны изменяться его текущие координаты. Поэтому для вычисления нового положения игрока, в зависимости от его текущих координат и направления движения, используется хэш %direction. Например, при перемещении на "север" необходимо вычесть 1 из координаты x игрока, а его координату y оставить без изменений. При движении на "восток" координата x остается без изменений, а координата y увеличивается на 1. (Начало координат находится в левом верхнем углу лабиринта. Ось x направлена вниз, а ось y — вправо.) Изменение координат выполняется в строках *33* и *34* программы.

- *Строки 12—15.* В этих строках с помощью оператора my описываются переменные, используемые в программе. Явного описания переменных требует оператор use strict. Текущее положение игрока хранится в переменных $cur_x и $cur_y, а его начальная позиция равна (0,0). Конечное положение игрока (3,3) хранится в переменных $x и $y.

- *Строка 17.* Эта подпрограмма отображает возможные направления движения игрока в зависимости от его положения (координат x и y).

- *Строка 20.* Из массива описания лабиринта @maze выбираются коды разрешенных направлений движения для текущей комнаты ($maze[$cx][$cy]). Затем из кода направления в цикле выделяются буквы, соответствующие сторонам света (с, о, з, в), полное описание которых хранится в хэше %full. Этот хэш используется только для преобразования кодов направлений (с) в название стороны света (север) и отображения его на экране монитора.

- *Строка 25.* Функции move_to передаются код направления (он сохраняется в переменной $new) и ссылки на текущие координаты игрока.

- *Строка 28.* Код направления преобразовывается к нижнему регистру с помощью функции lc, а функция substr выделяет первый символ из введенного пользователем кода направления. Результат снова присваивается переменной $new. Например, если пользователь введет Восток, переменной $new присваивается значение в, Запад — з и т.д.

- *Строка 29.* Введенный пользователем код сравнивается с кодами возможного направления движения из текущей комнаты ($maze[$$xref][$$yref]). При несовпадении выводится сообщение об ошибке.

- *Строки 33—34.* Выполняется изменение текущих координат x и y пользователя. Например, если был введен код направления в, из элемента хэша $direction{в} выбирается ссылка на двухэлементный массив (0, 1). В результате к текущей координате x прибавляется значение 0 — $direction{в}[0], а значение координаты y увеличивается на 1 — $direction{B}[l].

- *Строка 37.* В этой строке начинается тело основной программы. Цикл продолжается до тех пор, пока текущие координаты пользователя ($curr_x и $curr_y) не сравняются с координатами $x и $y секретной комнаты.

- *Строка 38.* Отображается "план" текущей комнаты.

- *Строки 39—41.* Введенный пользователем код направления движения считывается в переменную $move, после чего с помощью функции chomp

из нее удаляется символ перевода строки. Если пользователь введет символ q, игра завершается.

- *Строка 42.* Вызывается подпрограмма move_to, которой передаются код направления движения и ссылки на текущие координаты игрока. Эта подпрограмма выполняет пересчет текущих координат игрока $curr_x и $curr_y в зависимости от введенного кода направления.

Чтобы задать другую конфигурацию лабиринта, просто измените значения элементов массива @maze. Обратите внимание, что форма лабиринта не обязательно должна быть квадратной. Не обязательно также определять возможные направления движения из каждой комнаты, как и не обязательно, чтобы вообще существовал выход из лабиринта. Однако в комнатах не должно существовать дверей, ведущих наружу лабиринта. Учтите, что в программе не проверяется правильность построения лабиринта, хотя Perl выводит предупреждения, если вы неправильно зададите значения элементов массива @maze. Координаты секретной комнаты хранятся в переменных $x и $y. Измените их, если хотите переместить конечную точку лабиринта в другое место.

Резюме

На этом занятии речь шла об основных приемах работы со ссылками. Сначала вы познакомились с процессом создания ссылок на основные структуры данных Perl: скаляры, массивы и хэши, а затем рассмотрели способы работы с данными, определяемыми ссылками. Вы узнали также, как создавать ссылки на хэши и массивы, которым не назначены имена. В Perl подобные данные называются анонимными. И в конце занятия были рассмотрены способы создания сложных структур данных с помощью ссылок и приведены соответствующие примеры.

Вопросы и ответы

При попытке распечатать список списков с помощью оператора print "@LOL" выводятся значения типа ARRAY(0x101210), ARRAY(0x101400) и т.д. Почему это происходит?

В случае обычного массива оператор print @array выводит на печать его элементы, разделенные пробелом. Оператор print "@LOL" работает точно так же — он распечатывает элементы массива @LOL, которые являются ссылками на другие массивы. Чтобы распечатать элементы каждого из массивов, на который указывают элементы массива @LOL, воспользуйтесь методикой, описанной выше в разделе "Список списков, или двумерный массив" этого занятия.

Я пытаюсь создать ссылку на список с помощью оператора $ref=\($a, $Ь, $c). Почему в результате в переменной $ref оказывается ссылка на скаляр, а не на список?

В Perl конструкция \($a, $b, $c) является сокращенной записью списка (\$a, \$Ь, \$c)! Поэтому в переменной $ref окажется ссылка на последний элемент списка, находящийся в круглых скобках, т.е. $c. А чтобы создать ссылку на анонимный массив, воспользуйтесь оператором $ref=[$a, $b, $c].

Семинар

Контрольные вопросы

1. Чему будет равно значение переменной Sref после выполнения оператора `$ref=\"орехи";`?

   а) ничему, поскольку такой синтаксис недопустим;

   б) "орехи";

   в) ссылке на анонимный скаляр.

2. Что будет создано в результате объявления такой структуры?

```
$a=[
    {name==> "Иванов",kids==> [qw(Петя, Вася, Света) ]},
    { name==> "Петров",kids==> [qw(Саша, Маша,Натама)  ]},
];
```

   а) хэш, элементы которого являются хэшами, содержащими списки;

   б) список хэшей, содержащих список;

   в) список списков, содержащие другие списки.

Ответы

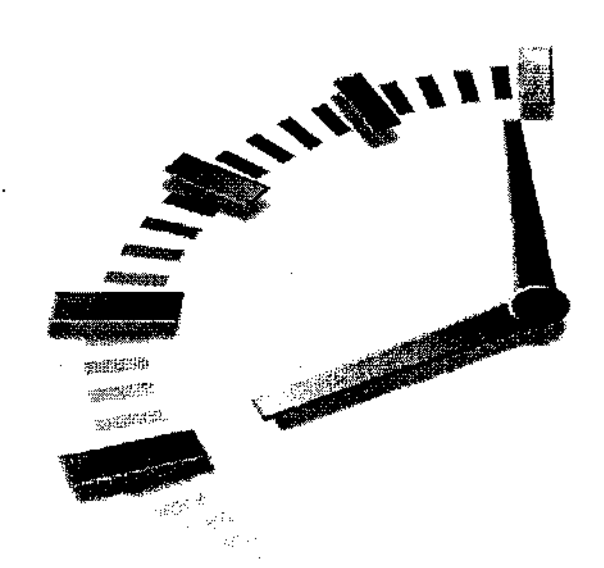
1. Правильный ответ — вариант в). Ссылки можно создать на любое значение, а не только на скаляр, массив или хэш. Например, с помощью оператора $ref=\100; создается ссылка на число. Если вы не уверены в ответе или ответили неправильно, попытайтесь создать короткую программу, выполнить ее под отладчиком и посмотреть, что же получится на самом деле.

2. Правильный ответ — вариант б). На этом занятии мы явно не рассматривали подобную структуру, однако вы должны легко понять, что она означает, — это список хэшей, элементы которых содержат списки (данные для ключа kids).

Упражнения

- Измените игру в лабиринт Maze так, чтобы появилась возможность движения игрока по диагонали. Для этого введите четыре новых кода направления, поскольку программа не позволяет идентифицировать коды `св`, сз, `юв` и юз. *Подсказка:* решение поставленной задачи состоит в помещении новых кодов направления в массив `@maze`, их описания в хэш `%full` и соответствующих числовых пар ([1,1], [-1,-1] и т.д.) в хэш `%direction`.

- Создайте структуру (можно на листе бумаги), описывающую счет за телефонные переговоры. В структуре (хэше) должны содержаться ключи и соответствующие для них данные (фамилия, адрес, номер телефона), а также список телефонных звонков, сделанных абонентом. В каждом элементе списка (тоже хэша) должна храниться дата и номер телефона.



14-й час

Использование модулей

Perl, как вы, наверное, уже заметили, чрезвычайно гибкий, можно сказать, универсальный, язык. Он позволяет работать с файлами, текстом, математическими формулами, алгоритмами и другими элементами, которые обычно присутствуют в любом языке программирования. В Perl большое внимание уделяется функциям специального назначения. Основу языка составляют регулярные выражения. Они очень важны для той области, в которой используется Perl, хотя многие языки прекрасно обходятся и без них. В Perl предусмотрена возможность запуска внешних программ (с помощью обратных кавычек, каналов и функции `system`), которую мы рассматривали на 11-м занятии, "Взаимодействие с операционной системой". Но опять-таки заметим, что во многих языках таких возможностей нет вообще.

При разработке любого языка программирования возникает искушение включить максимум полезных функций в основную часть самого языка. Если поддаться этому искушению, можно создать слишком громоздкий, неповоротливый язык, пользоваться которым будет неудобно. Например, некоторые разработчики языков считают, что в основу языка необходимо включить поддержку доступа к World Wide Web. Это, конечно, хорошая идея, но данная функция нужна далеко не каждому. И если через 10 лет Web будет иметь меньшее значение, чем сегодня, то придется принять решение удалить эту поддержку, в результате чего множество уже написанных программ попросту перестанет работать.

Разработчики Perl выбрали другой путь. Начиная с Perl 5, сам язык можно расширять путем использования *модулей*. Модули — это наборы программ на Perl, расширяющие возможности самого языка и область его применения. Например, существуют модули, реализующие возможности Web-броузеров, функции работы с графикой, поддерживающие функции Windows OLE, обеспечивающие возможность работы с базами данных и многое другое, что только можно вообразить. Однако следует отметить, что для работы самого интерпретатора Perl дополнительные модули не нужны. Он является полностью функциональным и законченным языком и не нуждается в дополнительных модулях.

С помощью модулей можно получить доступ к большой библиотеке отлаженных подпрограмм, которые будут полезны вам при написании собственных программ. Фактически, оставшаяся часть этой книги будет посвящена написанию CGI-программ с помощью модулей Perl.

В момент написания этой книги существовало свыше 3500 модулей, причем в стандартную поставку Perl включено не многим более 20 из них. Эти модули можно использовать в своих программах практически для любой цели, причем, по большей части, совершенно бесплатно. Не забывайте, что многие сложные проблемы, которые вам предстоит решить, на самом деле уже кем-то решены. И все, что нужно сделать, — это установить в системе нужный модуль и правильно им воспользоваться.

Основные темы этого занятия.

- Использование модулей в программах на Perl.
- Краткий обзор некоторых встроенных модулей.
- Описание списка основных модулей, входящих в поставку Perl.

Немного введения

Чтобы использовать некоторый модуль в программе на Perl, воспользуйтесь директивой use. Например, чтобы включить в программу модуль Cwd, просто поместите в любом месте своего кода следующий оператор:

```
use Cwd
```

Как уже было сказано, не имеет значения, в каком месте кода вы вставите конструкцию use Cwd. Но для ясности и простоты использования программы лучше всего поместить этот код в ее начало.

Модуль Cwd мы уже рассматривали на 10-м занятии, "Файлы и каталоги". Но в то время вы еще не знали, как он работает. При запуске программы, в которую включен код use Cwd, на самом деле происходит следующее.

1. Интерпретатор Perl открывает вашу программу и считывает весь код до тех пор, пока не найдет оператор use Cwd.
2. При установке интерпретатора Perl назначается определенный каталог, в котором должны храниться модули. В этом каталоге Perl и будет проводиться поиск модуля с именем Cwd. Этот модуль представляет собой файл, содержащий программу на языке Perl.
3. Perl считывает модуль, при этом инициализируются все функции и переменные, необходимые для работы этого модуля.
4. Интерпретатор Perl продолжает чтение и компиляцию программы с того места, где он прервался на обработку директивы use.

И это все. После того как Perl прочитает всю программу целиком и будет готов запустить ее, все функции, представляемые данным модулем, будут готовы к использованию.



> Вы, возможно, заметили, что конструкция use strict во многом напоминает use Cwd. Чтобы вы не запутались, нужно сказать следующее: оператор use — это команда общего назначения, "приказывающая" интерпретатору Perl сделать что-либо. Если говорить о use strict, то этот оператор ужесточает контроль интерпретатора использованием необъявленных и не инициализируемых переменных; модуля с именем strict не существует. А оператор use Cwd используется для включения некоторого модуля в программу. Пусть это отличие не слишком вас беспокоит — оно незначительное и вряд ли серьезно повлияет на вашу работу.

Когда вы включаете в программу оператор use Cwd, вам становится доступна новая функция cwd. Эта функция возвращает имя текущего рабочего каталога.

Чтение документации

Все модули Perl сопровождаются собственной документацией. Фактически, если вам доступен модуль, то доступна и документация на него, поскольку она, как правило, находится в самом модуле.

Чтобы просмотреть документацию к модулю, воспользуйтесь программой perldoc, указав имя этого модуля. Например, чтобы просмотреть документацию для cwd, в командном приглашении операционной системы просто наберите следующее:

```
perldoc Cwd
```

В результате документация будет отображаться в постраничном режиме. Ниже приведен пример (с некоторыми сокращениями).

```
Cwd(3)        13/Oct/98 (perl 5.005, patch 02)        Cwd(3)

   NAME
        getcwd - get pathname of current working directory

   SYNOPSIS
            use Cwd;
            $dir = cwd;

            use Cwd;
            $dir = getcwd;

            use Cwd;
            $dir = fastgetcwd;

DESCRIPTION
        The getcwd() function re-implements the getcwd(3) (or
        getwd(3)) functions in Perl.

        The abs_path() function takes a single argument and returns
        the absolute pathname for that argument. It uses the same
        algorithm as getcwd(). (actually getcwd() is abs_path("."))
```

Как следует из описания, модуль Cwd на самом деле позволяет использовать три новых функции: cwd, getcwd и fastgetcwd. Если они действительно вам нужны, то не пожалейте времени на ознакомление с документацией к модулю Cwd.



> Если вас интересует, как работает конкретный модуль, ознакомьтесь с его содержимым. В основном модули написаны на Perl и хранятся в системном каталоге интерпретатора. Например, модуль Cwd хранится в файле с именем `Cwd.pm`. Расположение этого файла может меняться, но обычно он находится в том каталоге, в котором установлен Perl. Путь к каталогу с модулями хранится в переменной `@INC`. Чтобы вывести ее значение, наберите после приглашения команду `perl -v`.

Поскольку многие модули созданы сторонними программистами, а не разработчиками Perl, и обычно распространяются бесплатно, то и качество документации на них может быть самым разным — от очень хорошего до крайне неудовлетворительного. Однако стоит отметить, что основные модули, входящие в стандартную поставку и упоминаемые в этой книге, а также другие известные модули, такие как Tk и LWP,

имеют очень хорошую документацию. Документация к менее популярным модулям обычно является довольно точной и аккуратной, но может оказаться и недостаточно подробной. Если вы не можете разобраться в работе некоторого модуля, обратитесь к материалу 16-го занятия, "Сообщество Perl", чтобы выяснить, какими ресурсами можно воспользоваться или как связаться с автором модуля, чтобы задать ему интересующий вас вопрос.

Какие могут возникнуть проблемы?

ЕСЛИ ваша версия интерпретатора Perl установлена правильно и не имеет внешних повреждений, то никаких проблем возникнуть не должно. Но, к сожалению, реальный мир далеко не идеален, и иногда что-то идет не так, как надо.

Если вы получили следующее сообщение об ошибке:

```
syntax error in file XXXX at line YYY, next two tokens "use Cwd"
```

то следует проверить, какая версия Perl у вас установлена. Попробуйте ввести после приглашения системы следующую команду:

```
perl -v
```

Если указанный Perl номер версии окажется меньше 5 — например, 4.036, — то у вас очень, очень старая версия Perl, и вы должны модернизировать ее. В ней нет многих возможностей версии Perl 5. Кроме того, во всех старых программах не обеспечивается должным образом защита данных. Фактически ни один из примеров, приведенных на 13-м занятии, "Структуры и ссылки", не будет работать в версии Perl 4. Если вы используете Perl 4, то уже должны были это заметить. Немедленно проведите модернизацию.

Вы можете получить также следующее сообщение об ошибке:

```
Can't locate Cwd.pm in @INC (@INC contains: path.. .path.. .path...)
BEGIN failed--compilation aborted
```

Обычно это означает одно из трех.

- Вы неправильно указали имя модуля (например, сделали в нем орфографическую ошибку).

  Имена модулей являются зависимыми от регистра, т.е. записи use Cwd и use cwd — это не одно и то же. У некоторых имен модулей внутри есть двоеточия (::) — например File: :Find, такие имена нужно тоже правильно набирать.

- Модуль, который вы пытаетесь подключить, не входит в стандартную поставку и не установлен на данном компьютере в соответствующем каталоге.

  В каждую версию Perl входит примерно 150 модулей — это и есть "стандартная поставка". Некоторые из них будут перечислены ниже на этом занятии. И все они должны работать без проблем. Если нужный вам модуль не входит в стандартную поставку, то вы или ваш системный администратор должны установить его.

  В, приложении "Инсталляция модулей в Perl" приводится инструкция о том, как установить эти дополнительные модули.

- Установка Perl проведена не до конца, либо интерпретатор скомпилирован с ошибками.

  К сожалению, время от времени это случается.

Интерпретатор Perl ищет установленные модули в тех каталогах, которые перечислены в сообщении об ошибке в переменной @INC. Если модули были перемешены, удалены или стали недоступны по какой-то другой причине, то самый простой выход — переустановить Perl. Но прежде чем взять на себя этот труд, нужно убедиться в том, что поврежденный модуль является стандартным. Любые дополнительные модули, которые вы установили самостоятельно, вполне могут оказаться в других каталогах, и это обычная практика. Более подробная информация о том, как устанавливать модули в нестандартных каталогах и как их использовать, содержится в приложении.

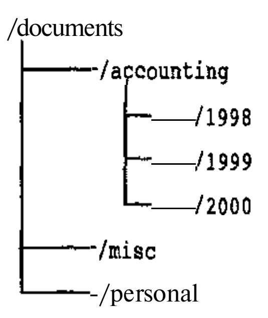
Краткий обзор

А теперь перейдем к краткому рассмотрению некоторых модулей, входящих в стандартную поставку Perl и уже установленных на вашем компьютере.

Исследование файлов и каталогов

На 10-м занятии, "Файлы и каталоги", вы узнали, как открывать каталоги и читать списки имен файлов, содержащихся в этих каталогах. Тогда же попутно встал вопрос о чтении подкаталогов, но мы не стали углубляться в него. Теперь настало время узнать, как без особых усилий выполнить рекурсивное чтение каталога и его вложенных подкаталогов.

Можно написать простую программу поиска файла, о котором неизвестно точно, в каком каталоге он находится. Например, вам нужно найти файл с именем `important.doc`, который находится в одном из подкаталогов каталога documents, имеющего следующую структуру:



На этом рисунке показана структура каталогов, которые находятся в родительском каталоге с именем documents. Найти файл, расположенный в одном из подкаталогов каталога documents, с помощью команд opendir/readdir/closedir непросто. Сначала нужно провести поиск по каталогу documents. Затем нужно продолжить поиск по всем подкаталогам, входящим в documents, — accounting, misc и personal, а потом по всем каталогам, содержащимся в этих каталогах, и т.д.

Это старая проблема, которую программисты решают снова и снова на протяжении последних 30 лет. Писать собственное решение данной проблемы — значит зря терять время. Как и следовало ожидать, разработчики Perl нашли простое решение, реализованное в модуле `File::Find`. Чтобы использовать этот модуль в программе, просто поместите в нее (желательно, в начале) следующий код:

```
use File::Find;
```

В результате вам станет доступна новая функция с именем `find`. Ее синтаксис выглядит следующим образом:

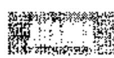
find *subref, dirlist*

Второй аргумент данной функции — это список каталогов, по которым проводится поиск. А первый аргумент — новый для вас; это *ссылка на подпрограмму.* Ссылка на подпрограмму создается точно так же, как и ссылка на скаляр или массив: это просто имя подпрограммы, перед которым стоит обратная косая черта. Чтобы сделать ссылку на подпрограмму, перед ее именем вы *должны* поставить символ &. Подпрограмма, имя которой указано в качестве первого параметра в функции find, будет вызываться для каждого файла и каталога из списка dirlist.

В листинге 14.1 приведена программа поиска пропавшего файла important.doc.

Листинг 14.1. Поиск файла

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:  use File::Find;
4:
5:  sub wanted {
6:          if ($_ eq "important.doc") {
7:                  print $File::Find::name;
8:          }
9:  }
10: find \&wanted, '/documents';
```

 Проведем анализ программы.

- *Строки 1—2.* С этих двух строк начинается практически любая программа на Perl. Ключ -w активизирует режим вывода предупреждений, а оператор use strict используется для ужесточения контроля интерпретатора Perl над ошибками в программе и выявления плохого стиля программирования.
- *Строка 3.* К вашей программе подключается модуль File::Find. В результате становится доступной функция find.
- *Строка 5.* Эта функция вызывается для каждого файла и каждого подкаталога, находящихся в каталоге /documents. Например, если в этом каталоге содержится 100 файлов и 12 подкаталогов, то подпрограмма wanted будет вызываться 112 раз.
- *Строка 6.* Когда вызывается функция wanted(), переменная $File: :Find: :name будет содержать полный путь к текущему обрабатываемому файлу, а переменная $_ — только имя файла. В этой строке определяется, является ли искомым именем файла important.doc; и если да, то печатается полный путь к этому файлу.
- *Строка 10.* При вызове функции find ей передается ссылка на подпрограмму Uwanted и имя каталога, в котором производится поиск файла. Функция wanted () вызывается для каждого файла и каталога, содержащихся в /documents.

Для функции, вызываемой find, будут доступны следующие переменные.

- $File::Find::name — полный путь к текущему файлу (каталог и имя файла);
- $File: :Find: :dir — имя текущего каталога;

- $_ — имя текущего файла (без указания каталога). Очень важно, чтобы вы не меняли значение переменной $_ в своей функции. Если же вы сделали это, то не забудьте в конце произвести обратную замену.

В листинге 14.2 приведен еще один пример использования модуля `File::Find`. В этом примере удаляются все файлы с расширением `.tmp` с устройств C: и D:. Эти временные файлы имеют свойство накапливаться и "съедать" свободное пространство жесткого диска. Данную программу можно легко адаптировать для удаления файлов на компьютере, на котором установлена система UNIX, или для выполнения любых других функций по обслуживанию файловой системы.

Листинг 14.2. Удаление временных файлов

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:  use File::Find;
4:
5:  sub wanted {
6:    # Проверим, что это: имя файла или каталога
7:    if ( -f $File::Find::name ) {
8:          # В пути к файлу должна быть строка ".tmp"
9:          if ( $File::Find::name=~/\.tmp$/i ) {
10:                 print "Удаление $File::Find::name";
11:                    unlink $File::Find::name;
12:             }
13:    }
14: }
15:  find(\&wanted, 'c:/', 'd:/');
```

АНАЛИЗ Программа, приведенная в листинге 14.2, в основном аналогична программе из листинга 14.1. Проведем анализ ее отличий.

- *Строка 7.* Проводится проверка файла, имя которого передано подпрограмме wanted, для выяснения, обычный ли это файл или каталог. Как вы помните, данная подпрограмма вызывается и для файлов, и для каталогов.
- *Строки 9–11.* Проводится проверка имени файла для выяснения того, содержится ли в конце его расширение `.tmp`. И если да, то данный файл удаляется с помощью команды unlink.

Копирование файлов

Еще одну распространенную задачу — копирование файлов — можно выполнить в Perl довольно сложным путем.

1. Откройте исходный файл для чтения.
2. Откройте выходной файл для записи.
3. Прочитайте исходный файл и выполните запись его содержимого в выходной файл.
4. Закройте оба файла.

И, конечно, после каждого шага вы должны убедиться в том, что никаких ошибок не произошло и что каждая операция записи была выполнена успешно, А теперь позвольте мне показать вам более простой способ. В Perl предусмотрен модуль File::Copy, осуществляющий копирование файлов. Ниже приведен пример использования этого модуля.

```
use File::Copy;
copy("sourcefile", "destination") ||
      warn "Ошибка при копировании файлов: $!";
```

Этот фрагмент кода копирует содержимое исходного файла sourcefile в выходной файл destination. Функция copy возвращает значение 1 в случае успешного завершения операции и 0, если возникла какая-то проблема, При этом переменной $! присваивается текст соответствующего сообщения об ошибке.

В модуле File::Copy предусмотрена также функция move. Путем простого изменения структуры элементов каталога файловой системы функция move пытается выполнить операцию перемещения файлов без физического переноса их содержимого. Если исходный и выходной файлы расположены в одном каталоге и имеют разные имена, выполняется обычная операция переименования файлов. Обычно функция move так работает в случае, когда оба файла находятся в одной файловой системе или на одном диске. Если же по какой-либо причине выполнить быструю операцию перемещения файлов не удается, функция move сначала копирует исходный файл в выходной, а затем удаляет первоначальный файл. Рассмотрим следующий пример:

```
use File::Copy;
if (not move("important.doc", "d:/archives/documents/important.doc")) {
   warn "Ошибка при перемещении файла important.doc: $!";
   unlink "d:/archives/documents/important.doc";
>
```

Данный фрагмент кода перемешает файл important.doc из его текущего каталога в каталог d:/archives/documents. Если при выполнении функции move произошел сбой, то в выходном каталоге возможно появление частично скопированного файла important.doc. В случае неудачного завершения операции move функция unlink удаляет частично скопированный файл из выходного каталога.

Ау! Есть здесь кто-нибудь?

Модули Perl не ограничиваются только выполнением операций над файлами и каталогами. Например, модуль Net::Ping можно использовать для определения того, может ли компьютер нормально взаимодействовать с другим узлом сети.

Имя модуля Net: :Ping происходит от утилиты ping системы UNIX. Эта утилита получила свое имя от слова "ping", обозначающего акустический импульсный сигнал, который используется на подводных лодках для обнаружения объектов по принципу отражения звука. Утилита ping посылает некоторый пакет другому компьютеру в сети. Если этот компьютер включен и нормально функционирует, то он посылает ответ, и команда ping сообщает об успешном выполнении операции. Модуль Net::Ping, пример использования которого приведен ниже, работает аналогично.

```
use Net::Ping;

if ( pingecho("www.yahoo.com", 15) ) {
    print "Сервер Yahoo функционирует нормально.";
} else {
    print "С Yahoo что-то произошло...";
}
```

Как видно из данного фрагмента кода, в модуле Net: :Ping предусмотрена функция с именем **pingecho**. У этой функции два аргумента. Первый — это имя узла сети, который нужно проверить на работоспособность (в данном случае — **www.yahoo.com)**. Второй аргумент указывает, как долго (в секундах) функция pingecho должна ждать ответа.



> Из-за особенностей языка Perl для систем Windows 95/98/NT к моменту написания данной книги (лето 1999) модуль Net: :Ping не работал. Этот модуль зависит от функции alarm, которая не работает в системе Windows. Однако фирма Activestate — главный разработчик Perl для системы Windows — объявила о планах реализации многих недостающих функций для Windows и о внесении соответствующих изменений в Perl.

Еще раз, пожалуйста, но по-английски!

Модуль English позволяет обращаться к некоторым специальным переменным Perl по их более понятным именам, как показано в следующем примере:

```
use English;

while(<>) {
    print $ARG;
}
```

В приведенном фрагменте кода конструкция while{<>) обычно считывает одну строку текста из потока STDIN и присваивает его переменной $_. В нашем примере по-прежнему все так и происходит. Но при использовании оператора use English к переменной **$_** можно обращаться по имени **$ARG**. Ниже приведен частичный список специальных переменных и их английских эквивалентов.

Специальная переменная	Английское название
$_	$ARG
@_	@ARG
$!	$OS_ERROR
$^O	$OSNAME
$0	$PROGRAM_NAME

Полный список специальных переменных и их английских эквивалентов можно найти в электронной документации к модулю English.

Дополнительные средства диагностики

Модуль diagnostics языка Perl помогает находить ошибки в программе. Если по ходу чтения вы будете учиться языку, набирая примеры из данной книги, то наверняка будете получать от интерпретатора Perl сообщения об ошибках, которые не сможете понять до конца. Например, короткая программа

```
#!/usr/bin/perl -w

use strict;
print "Для получения помощи отправьте сообщение по адресу help@support.org\n";
```

заставит Perl выдать следующее предупреждающее сообщение:

```
In string, @support now must be written as \@support at line 4
Global symbol "@support" requires explicit package name at line 4
```

Благодаря модулю diagnostics Perl выдает подробные предупреждения и сообщения об ошибках. Можно изменить приведенную выше программу-образец, включив в нее модуль диагностики следующим образом:

```
#!/usr/bin/perl -w

use strict;
use diagnostics;

print "Для получения помощи отправьте сообщение по адресу help@support.org\n";
```

В результате такого изменения программы и использования модуля диагностики будет выдано более подробное сообщение:

```
In string, @support now must be written as \@support at line 4
Global symbol "@support" requires explicit package name at ./diag.pl line 5 (#1)

   (F) You've said "use strict vars", which indicates that all variables
  must either be lexically scoped (using "my"), or explicitly qualified to
   say which package the global variable is in (using "::").
```

Если вы немного поразмышляете об этих двух сообщениях, то станет ясно, что они связаны. Первое сообщение очевидно. Perl говорит о том, что направлять письма по электронной почте нужно по адресу help\@support.com. Теперь, после данного объяснения, второе сообщение становится более понятным. Так как была активизирована команда use strict, переменную @support следовало объявлять с помощью my. Но дело в том, что (Isupport — не переменная, а часть электронного адреса, которая просто была неправильно интерпретирована Perl.

Буква перед сообщением указывает на тип ошибки. (W) обозначает предупреждение, (D) говорит об использовании устаревшего и не рекомендуемого синтаксиса, (S) — это строгое предупреждение, а (F) — это серьезная ошибка. Программа на Perl продолжает работу при выдаче всех типов сообщений, за исключением (F).

В документации по Perl предусмотрено около 60 страниц описания сообщений об ошибках. Если вам трудно понять, что означают краткие сообщения Perl об ошибках, то иногда разобраться в них поможет команда use diagnostics.



Полный список сообщений об ошибках и диагностической информации можно найти в разделе perldiag электронной документации по Perl.

ПОЛНЫЙ СПИСОК стандартных модулей

Пространное описание абсолютно всех модулей, включенных в Perl, выходит за рамки данной книги. Ниже перечислены модули, входящие в стандартную поставку Perl, и дано их краткое описание. Если вы хотите знать, какие операции выполняет модуль и как он работает, используйте утилиту perldoc, чтобы просмотреть документацию к данному модулю.

Имя модуля	Описание
AutoLoader	Позволяет Perl компилировать функции только по мере необходимости
AutoSplit	Разделяет модули для автозагрузки
Benchmark	Позволяет многократно замерять скорость выполнения функций Perl для проведения тестирования производительности программы
CGI	Разрешает доступ к функциям CGI при программировании сценариев для Web-серверов, о которых пойдет речь в части III, "CGI-программирование на Perl", данной книги
CPAN	Обеспечивает доступ к архивам модулей Perl для инсталляции новых модулей
Carp	Генерирует сообщения об ошибках
DirHandle	Обеспечивает объектный интерфейс к дескрипторам каталогов
Env	Создает связь между переменными окружения операционной системы и переменными языка Perl
Exporter	Позволяет написать собственные модули
ExtUtils::*	Позволяет написать собственные модули или установить имеющиеся
File::*	Предлагает дополнительные операции с файлами, такие как File::Copy
File::Spec::*	Позволяет выполнять с именами файлов операции, независимые от операционной системы
FileCache	Открывает больше файлов, чем обычно позволяет операционная система
FindBin	Находит имя текущей выполняющейся программы
Getopt::*	Позволяет обрабатывать в программе параметры командной строки
I18N::Collate	Позволяет выполнять сортировку в соответствии с определенным алфавитом
IPC::*	Обеспечивает взаимодействие между процессами, например двух- или трехуровневый конвейер
Math::*	Позволяет использовать расширенные математические библиотеки для выполнения операций с произвольной точностью над целыми, комплексными числами и числами с плавающей точкой
Net::*	Позволяет получать информацию об узлах сети. Например, Net::hostent преобразует IP-адреса, такие как 204.71.200.68, в имена узлов сети, например www.yahoo.com
Pod::*	Обеспечивает доступ к программам форматирования в стиле старой документации Perl
Symbol	Позволяет просматривать или изменять собственную таблицу символов Perl
Sys::Hostname	Возвращает имя вашего компьютера в сети, соответствующее его IP-адресу
Sys::Syslog	Позволяет сделать запись в журнале ошибок системы UNIX

Имя модуля	Описание
Term::*	Обеспечивает интерфейс функций управления терминалом для позиционирования курсора, очистки экрана и т.д.
Text::Abbrev	Строит таблицы сокращений
Text::ParseWords	Позволяет анализировать текст по словам
Text::Soundex	Классифицирует слова на основе произношения с помощью метода Soundex
Tie::*	Связывает переменные Perl с функциями, чтобы вы могли реализовать собственные массивы и хэши
Time::*	Позволяет выполнять различные операции с датами и временем. Например, можно преобразовать дату, заданную в формате Sat Jul 24 16:21:38 EDT 2000, в количество секунд, прошедших с 0 часов 1 января 1970 года
constant	Позволяет определить постоянные значения
integer	В некоторых случаях заставляет Perl выполнять математические действия над целыми числами, а не над числами с плавающей точкой
locale	Заставляет выполнять сравнение строк на основе установленного алфавита

Что дальше

ЕСЛИ ВЫ хотите получить представление о том, какие модули можно получить в свое распоряжение, причем бесплатно, воспользуйтесь Web-броузером и обратитесь по адресу http://www.cpan.org. Представленные там модули разбиты по категориям.

Для установки некоторых модулей требуется компилятор языка С и минимальная среда разработки. Этих средств может не быть на компьютере, на котором установлена система Windows. В версии Perl фирмы Activestate содержится утилита PPM, которую можно использовать для просмотра и инсталляции новых модулей.

В приложении содержатся пошаговые инструкции по инсталляции модулей на компьютерах, под управлением систем UNIX и Windows. В этих инструкциях описывается, как использовать модуль CPAN (для системы UNIX) и утилиту PPM фирмы Activestate для инсталляции новых модулей.

Резюме

На этом занятии вы узнали, как использовать модули для расширения возможностей языка Perl. Это позволяет легко решать многие стандартные задачи. Описанный здесь универсальный модульный подход Perl будет использоваться на протяжении оставшейся части книги. Кроме того, вашему вниманию были представлены некоторые широко используемые модули, а также полный список модулей, включенных в стандартную поставку Perl.

Вопросы и ответы

Что означают двоеточия (::) в именах переменных в модуле File::Find, например $File::Find::dir?

Модули Perl могут создавать альтернативные области имён переменных, называемые *пространствами имен.* Это сделано для того, чтобы не путать имена глобальных переменных модуля и имена глобальных переменных вашей программы. Поэтому глобальная переменная в модуле Cwd будет называться $Cwd::x. Большинство глобальных переменных вашей программы на самом деле имеют полное имя, которое отличается от сокращенного. Например, полным именем переменной $x будет $main::x. Но пока для нас это не имеет особого значения.

На моем компьютере установлена система Windows 95/98/NT, и нужный мне модуль нельзя установить с помощью программы PPM Activestate. Как же мне инсталлировать его?

К сожалению, для установки большинства модулей CPAH необходима полная UNIX-подобная среда разработки для компиляции и инсталляции модулей; такую среду нелегко получить на компьютере, на котором установлена система Windows. Если вы хорошо владеете искусством работы с компилятором С, то можете загрузить среду разработки и создать собственный модуль; но сделать это не так просто.

У меня есть старая программа на Perl, в которой вместо оператора use используется require. Что такое require?

Оператор require аналогичен use. Поскольку в Perl 4 не было ключевого слова use, в нем использовалось require. Оператор require заставляет интерпретатор находить библиотечный файл и включает его в вашу программу — подобно use. Но главное отличие состоит в том, что директива require может обрабатываться во время выполнения программы, тогда как директива use выполняется только в процессе загрузки программы (т.е. во время компиляции).

Семинар

Контрольные вопросы

1. ЕСЛИ ВЫ хотите дважды использовать функцию cwd в программе, то сколько раз вы должны ввести команду use Cwd;?

   а) один раз;

   б) по одному разу для каждого случая использования функции cwd, т.е. всего два раза;

   в) ни разу, так как cwd — это встроенная функция.

2. В каком модуле предусмотрен псевдоним для переменной $_?

   а) LongVars

   б) English

   в) у $_ нет псевдонимов.

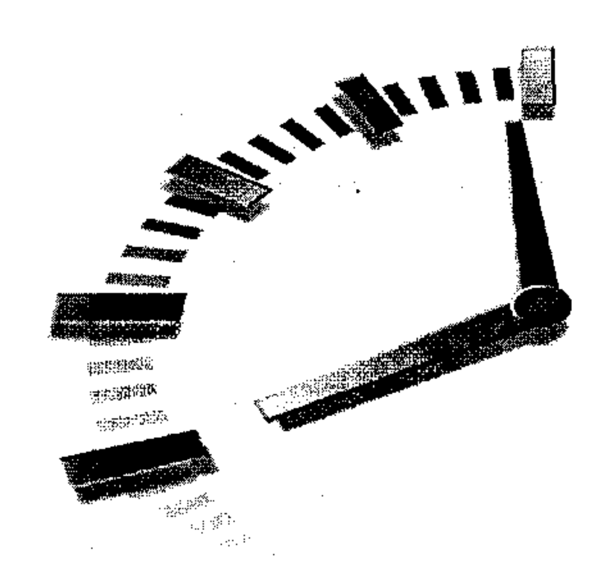
Ответы

1. Правильный ответ — вариант а). После того как модуль подключен к программе с помощью директивы use, все его функции будут доступны для использования в остальной части программы.

2. Правильный ответ — вариант б). Использование оператора use English приводит к тому, что к переменной $_ можно также обращаться под именем $ARG.

Упражнения

- Откройте приложение к этой книге и попытайтесь использовать приведенные в нем команды для инсталляции модуля Bundle::LWP из CPAN. Вам понадобится этот модуль для выполнения примеров, описанных на 24-м занятии, "Создание интерактивного Web-сервера".



15-й час

Обработка данных в Perl

До сих пор мы рассматривали программы, в которых исходные данные вводились в процессе диалога с пользователем или из файла, а результат вычислений отображался на терминале. Но вы когда-нибудь задумывались над вопросом: "Что происходит с данными, полученными в результате работы программы, после ее завершения?" Ничего особенного, они попросту теряются, будто ничего и не было. Таким образом, при следующем запуске программы нужно начинать все вычисления сначала.

Вот тут-то на помощь и приходят *базы данных.* Они используются для хранения данных, предназначенных для последующей обработки. Более того, правильно спроектированная база данных может использоваться многими программами для выполнения запросов, создания всевозможных отчетов и ввода данных. Перед разработкой базы данных вы должны тщательно продумать ее структуру и определить способы хранения данных. Есть еще одна немаловажная деталь, которую нужно иметь в виду, — метод доступа к данным: будет ли с базой данных работать один человек, или необходимо обеспечить доступ одновременно для нескольких пользователей.

На этом занятии мы рассмотрим несколько способов хранения данных, предназначенных для дальнейшего использования.

Основные темы этого занятия.

- Создание DBM-файлов и хранение данных в них.
- Использование в качестве базы данных обычного текстового файла.
- Произвольный доступ к данным, хранящимся в файлах.
- Блокировка данных в файле для предотвращения одновременного доступа к ним нескольких пользователей.

Файлы DBM

При программировании на Perl использование DBM-файлов является самым простым способом хранения структурированных данных. Файлы DBM обладают одним замечательным свойством — в программах на Perl их можно напрямую связать с хэ-

шем. При этом чтение и запись DBM-файлов сводится к простым операциям с хэшем, о которых шла речь на 7-м занятии, "Хэши".

Чтобы связать хэш с DBM-файлом, в Perl используется функция dbmopen, синтаксис которой выглядит так:

```
dbmopen(хэш, имя_файла, права_доступа)
```

В результате выполнения этой функции указанный вами *хэш* подключается к DBM-файлу. Параметр *имя_файла* на самом деле определяет два файла на жестком диске: *имя_файла*.pag и *имя_файла*.dir. Они используются Perl для хранения данных в иерархическом виде и быстрого доступа к ним. Эти файлы *не являются* текстовыми, поэтому их нельзя редактировать с помощью обычного текстового редактора. Кроме того, не обращайте внимания, если один из файлов имеет нулевую длину или его размер слишком велик по сравнению с сохраненными в нем данными. Это вполне нормальное явление.

Третий параметр функции dbmopen определяет права доступа, которые назначаются двум DBM-файлам при их создании. При работе в системе UNIX всегда используйте осмысленные значения *прав_доступа*. Это позволит контролировать доступ к вашим DBM-файлам. Например, значение кода прав доступа, равное 0666, обеспечивает доступ по чтению и записи к вашим DBM-файлам для всех пользователей данного компьютера; значение 0644 позволяет вам читать и записывать данные, в то время как для остальных пользователей обеспечивается только режим чтения. При работе в системе Windows данный параметр не играет особой роли, поскольку в ней не предусмотрены средства управления доступом. Поэтому всегда используйте значение 0666.

Функция dbmopen возвращает истинное значение, если операция подключения хэша к DBM-файлу прошла успешно. А теперь давайте рассмотрим пример:

```
dbmopen(%hash, "dbmfile", 0644) ||
        die "Ошибка при открытии DBM-файла: $!";
```

После выполнения этого оператора устанавливается связь хэша %hash с DBM-файлом dbmfile. Для хранения хэша на диске Perl создает два файла: dbmfile.pag и dbmfile.dir. Если в последующих операторах значение элементов хэша будет изменено (как показано ниже на примере), Perl автоматически обновит соответствующие DBM-файлы:

```
$hash{'кошачьи'}="кошка";
$hash{'собачьи'}="собака";
```

Обращение к элементам хэша автоматически приводит к считыванию информации из DBM-файла, например:

```
print $hash{'собачьи'};
```

Чтобы разорвать связь хэша %hash с DBM-файлом, используется функция dbmclose:

```
dbmclose(%hash);
```

После выполнения этой функции элементы хэша 'кошачьи' и 'собачьи' останутся в DBM-файле. В результате при следующем запуске программы и связывании хэша %hash с DBM-файлами значение указанных элементов хэша будет восстановлено.

С хэшами, связанными с DBM-файлами, можно выполнять те же операции, что и с обычными хэшами, например использовать функции keys, values и delete. Чтобы очистить хэш (и соответственно DBM-файл), присвойте ему пустой список, как показано ниже:

```
%hash=();
```

Чтобы инициализировать хэш и соответствующий ему DBM-файл, после выполнения функции dbmopen присвойте ей нужные значения в списке.

Некоторые важные замечания

В этом разделе мы приведем несколько полезных замечаний, которые нужно иметь в виду при связывании хэша с DBM-файлом.

- Ограничение на длину ключей и данных. Хотя в Perl не накладывается никаких специальных ограничений на длину ключей и данных хэша, тем не менее, при связывании с DBM-файлом суммарный размер одного ключа и хранимых в нем данных не должен превышать 1024 символа. Это ограничение обусловлено структурой DBM-файла. На общее число ключей в хэше не накладывается никаких ограничений, оно зависит только от типа используемой файловой системы.

- После выполнения функции dbmopen первоначальные значения элементов хэша теряются. Поэтому лучше всего для операции связывания выбирать чистый хэш. Рассмотрим следующий пример:

```
%h=();
$h{'одногорбые'}="верблюд";
dbmopen(%h, "dbmfile", 0644) ||
        die "Ошибка при открытии DBM-файла: $1";
print $h{'одногорбые'}; t Ничего не будет напечатано
dbmclose(%h);
```

  В этом примере после выполнения функции dbmopen ключ 'одногорбые' хэша %h теряется.

- После выполнения функции dbmclose значения ключей связанного хэша теряются. Вот пример:

```
dbmopen(%h, "dbmfile", 0644) ||
        die "Ошибка при открытии DBM-файла: $!";
$h{'парнокопытные'}="корова";
dbmclose(%h);
print $h{'парнокопытные'}; # Ничего не будет напечатано
```

  В этом примере новый ключ 'парнокопытные' будет добавлен к DBM-файлу. Однако после выполнения функции dbmclose все ключи связанного хэша теряются, а хэш %h полностью очищается.

Обработка больших DBM-файлов

Предположим, что некоторый хэш связан с DBM-файлом. Для определенности будем считать, что вы пишете на Perl программу, которая сохраняет в файле сведения о сотрудниках: фамилия, должность, номер телефона и др. Очевидно, что если сотрудников достаточно много, то через некоторое время ваш хэш станет очень большого размера. Причина заключается в том, что каждый раз при запуске программы ее значения восстанавливаются из DBM-файла, добавляются в хэш и снова записываются в файл при завершении программы. Таким образом, если вы не предпримете специальных действий, значения из вашего хэша никогда не будут автоматически удаляться.

Если DBM-файл, называемый records, имеет большое количество **информации,** то при выполнении приведенного ниже фрагмента программы могут возникнуть проблемы.

```
dbmopen(%recs, "record", 0644) J|
        die "Ошибка при открытии DBM-файла record: $1";
foreach my $key (keys %recs) {
   print " $key = $recs{$key}\n";
}
dbmclose(%recs);
```

Не ищите ошибку в коде, ее там нет! Вначале выполняется связывание хэша `%recs` с DBM-файлом, затем с помощью оператора keys `%recs` из него извлекается список всех ключей, после чего в цикле foreach my $key распечатывается ключ и соответствующее ему значение.

Если список ключей хэша `%recs` велик, оператор keys `%recs` может выполняться достаточно продолжительное время и завершиться аварийно из-за нехватки оперативной **памяти.** Поэтому в Perl предусмотрена еше одна функция, предназначенная для обработки элементов хэша по одному за раз. Она называется each. Ее синтаксис выглядит так:

($*ключ*, $ *значение)* = **each(%*хэш*);**

Функция each возвращает список, состоящий из двух элементов, — *ключа* и его ***значения,*** извлеченных из *хэша.* При каждом вызове этой функции она возвращает из хэша очередную пару **ключ—значение.** Если ключи в хэше исчерпаны, функция возвращает пустой список. Таким образом, приведенный выше фрагмент кода можно переписать так, чтобы с его помощью можно было обрабатывать хэши большого размера:

```
dbmopen(%recs, "record", 0644) ||
        die "Ошибка при открытии DBM-файла record: $!";
while( ($key, $value)=each %recs) {
   print " $key = $value\n";
}
dbmclose(%recs);
```



Функцию each можно использовать для перебора элементов любого хэша, а не только того, который связан с **DBM-файлом.**

Пример: программная реализация записной книжки

Теперь, после того как вы научились сохранять данные программы на диске, самое время найти полученным знаниям достойное применение. В этом примере мы рассмотрим программную реализацию электронной записной книжки. Программа называется memopad, а ее текст приведен в листинге 15.1. Информация в записной книжке хранится в виде хэша, что позволяет получить к ней быстрый доступ с помощью простых запросов. Пример сеанса работы с программой memopad приведен в листинге 15.2.

Для запроса к программе memopad вводится название темы с вопросительным знаком. Чтобы занести в программу новую информацию, наберите фразу в виде *X* is У, где *X* — название темы, а *Y* — информация, которая должна ассоциироваться с этой темой. Для поиска информации в базе данных используется запрос "like *шаблон?",* где

шаблон — это регулярное выражение, используемое для поиска нужной темы. В результате выполнения запроса программа выведет список всех тем, соответствующих шаблону. Для выхода из программы наберите quit. Благодаря использованию хэшей, связанных с DBM-файлами, вся вводимая в программу memopad информация будет запоминаться на диске и восстанавливаться при повторном запуске этой программы.

Листинг 15.1. Исходный текст программы memopad

```
#!/usr/local/perl -w
use strict;

my(%answers, $subject, $info, $pattern);

dbmopen(%answers, "answers", 0644) ||
      die "Ошибка при открытии DBM-файла 'answers': $!";
while(1) {
   print "Введите запрос или 'quit' для выхода: ";
     chomp($_=lc(<STDIN>));
    last if (/^quit$/);
   if (/like\s+(.*)\?/) {
        $pattern=$1;
       while( ($subject,$info)=each(%answers) ) {
              if ($subject=~/$pattern/) {
               print "Шаблон '$pattern' встретился в '$subject'\n";
               }
         }
    } elsif(/(.*)\?/) {
          $subject=$1;
         if ($answers{$subject}) {
           print "$subject - это $answers{$subject}\n";
          } else {
            print "Про $subject нам ничего не известно\n";
            }
    } elsif(/(.*)\sis\s(.*)/) {
          $subject=$1;
          $info=$2;
          $answers{$subject}=$info;
        print "Мы запомнили, что $subject - это $info\n";
    } else {
         print "Ошибочный запрос\n";
     }
}
dbmclose(%answers);
```

Листинг 15.2. Пример диалога с программой memopad

```
Введите запрос или 'quit' для выхода: perl?
Про perl нам ничего не известно
Введите запрос или 'quit' для выхода: perl is язык программирования
Мы запомнили, что perl - это язык программирования
Введите запрос или 'quit' для выхода: Web-сервер perl is http://www.perl.org
Мы запомнили, что web-сервер perl - это http://www.perl.org
```

```
Введите запрос или 'quit' для выхода: perl?
perl - это язык программирования
Введите запрос или 'quit' для выхода: like perl?
Шаблон 'perl' встретился в 'perl'
Шаблон 'perl' встретился в 'web-сервере perl'
Введите запрос или 'quit' для выхода: quit
```

 Проведем анализ программы.

- *Строки 1—2.* С этих двух строк начинается практически любая программа на Perl. Ключ -w активизирует режим вывода предупреждений, а оператор use strict используется для ужесточения контроля интерпретатора Perl над ошибками в программе и выявления плохого стиля программирования.
- *Строка 6.* Выполняется привязка хэша %answers к DBM-файлу answers с помощью функции dbmopen. В результате на диске создаются два файла — answers.pag и answers.dir.
- *Строка 7.* Оператор while(l) задает бесконечный цикл. Для завершения работы цикла и программы в теле цикла используется оператор last.
- *Строка 9.* Эта строка может вас сбить с толку, поскольку в ней выполняется сразу несколько операций. С помощью функции 1с введенная пользователем строка преобразуется к нижнему регистру. Поскольку оператор <STDIN> используется в скалярном контексте функции 1с, из стандартного устройства ввода читается одна строка, ее символы преобразуются к нижнему регистру и результат присваивается переменной $_. С помощью функции chomp из строки $_ удаляются символы перехода на новую строку.
- *Строка 10.* Если во входной строке содержится единственное слово quit, работа цикла while завершается.
- *Строка 11.* Если во входной строке (она находится в переменной $_) будет найдено слово like, после которого следует текст, заканчивающийся вопросительным знаком, то сам текст помещается в переменную $1 (в шаблоне используется группировка с помощью круглых скобок).
- *Строка 12.* Строка, помещенная оператором поиска по шаблону в переменную $1, сохраняется для дальнейшего использования в переменной $pattern.
- *Строки 13-17.* Выполняется последовательный просмотр всех ключей хэша %answers в поисках ключа, который соответствует строке, находящейся в переменной $pattern. По мере нахождения ключей они выводятся на печать.
- *Строка 18.* Эта строка является продолжением оператора if, начало которого находится в 11 строке. В ней проверяется, не содержится ли в конце введенной пользователем строки вопросительный знак. При соответствии шаблону часть строки до вопросительного знака сохраняется в переменной $1.
- *Строка 19.* Строка, помещенная оператором поиска по шаблону в переменную $1, сохраняется для дальнейшего использования в переменной Ssubject.
- *Строки 20—24.* Если в хэше %answers существует ключ, имя которого находится в переменной Ssubject, сам ключ и ассоциированные с ним данные

выводятся на печать. В противном случае программа выводит сообщение, что по указанной теме ей ничего не известно.

- *Строки 25–27.* Строка 25 является продолжением оператора if, начало которого находится в 11 строке. В ней проверяется, соответствует ли введенная строка формату *X* is *Y*. При положительном ответе часть *X* запоминается в переменной `$subject`, а часть `Y` — в $info.
- *Строка 28.* Информация, находящаяся в переменной `$info`, запоминается в хэше `%answers` в ключе, имя которого находится в переменной `$subject`.
- *Строка 34.* С помощью функции `dbmclose` разрывается связь хэша `%answers` с DBM-файлом.

Использование текстовых файлов в качестве базы данных

Часто бывает, что база данных имеет простую структуру и небольшой размер. В качестве примера можно привести список пользователей небольшого узла, список компьютеров в локальной сети, список адресов часто посещаемых Web-страниц или личную адресную книгу. Все эти вещи можно реализовать с помощью простых форм баз данных, для хранения которых вполне подойдут обычные текстовые файлы. Поэтому давайте рассмотрим все плюсы и минусы технологии, описанной в данном разделе.

Начнем с хорошего. Использование текстовых файлов для хранения баз данных имеет несколько неоспоримых преимуществ перед более сложными альтернативами, такими как DBM-файлы или системы управления большими базами данных типа Oracle и Sybase. Ниже перечислены некоторые из основных преимуществ текстовых файлов.

- Базы данных, хранящиеся в текстовых файлах, являются переносимыми. Их можно без всяких проблем использовать практически на любой компьютерной платформе.
- Текстовые файлы можно редактировать с помощью обычного текстового редактора, а также распечатать на бумаге без привлечения каких-либо специальных средств.
- Текстовые файлы баз данных очень просто создавать, а также вносить в них первоначальные данные.
- Текстовые файлы баз данных могут быть легко импортированы в программы электронных таблицы, текстовые процессоры или СУБД. Практически все известные приложения могут импортировать данные, хранящиеся в текстовых файлах.

А теперь, как вы могли догадаться, настала очередь поговорить о плохом. Чтобы разобраться в истоках проблемы, давайте рассмотрим традиционный метод организации баз данных в текстовых файлах. В каждой строке текстового файла обычно хранится одна запись, которая состоит из столбцов данных, называемых полями. Для операционной системы текстовый файл базы данных ничем не отличается от обычного файла — потока текстовых символов, разбитого на строки. Давайте рассмотрим пример простейшей текстовой базы данных.

```
Борис  555-1212
Мария  555-0912
```

Павел 555-0012
Ольга 555-1190

Этот файл базы данных хранится на диске в виде непрерывного потока символов:

Борис[пробел]555-1212[новая строка]Мария[пробел]555-0912[новая строка] ...

В этом потоке элементы [пробел] и [новая строка] представляют зависимые от конкретной операционной системы признаки пробела и новой строки. Например, в качестве признака новой строки в разных операционных системах может использоваться либо символ перевода строки, либо символ возврата каретки, либо их комбинация. Другими словами, символы всех полей и всех записей файла базы данных упакованы в один сплошной поток байтов файла. Правда, стоит отметить, что подобные файлы отображаются в текстовом редакторе, распечатываются на принтере и представляются Perl в удобном для восприятия человеком виде.

А теперь, после того как вы познакомились со структурой текстовых баз данных, давайте обсудим некоторые их недостатки.

- В середину текстового файла нельзя вставить новые данные. При вставке новых записей файл нужно полностью или частично обновлять. Поэтому вставка данных в начало или середину файла автоматически вызывает перезапись расположенных следом данных. Предположим, например, что после записи Борис 555-1212 необходимо вставить запись Сергей 555-613. В результате данные **Мария[пробел]555-0912[новая** строка] ... нужно сдвинуть к концу файла так, чтобы после записи Борис 555-1212 образовалось место для новой записи Сергей 555-613:

  Борис[пробел]555-1212[новая строка]Сергей[пробел]555-613[новая строка]
  Мария[пробел]555-0912[новая строка] ...

  Таким образом, видно, что вставка записей в середину текстовой базы данных — довольно медленная и не лишенная ошибок операция (особенно при больших размерах файлов). При сбое в момент перезаписи файла может произойти частичная или полная потеря данных.

- Приведенные выше замечания справедливы также и для операции удаления записей, которая является обратной вставке данных. Удалить данные из начала или середины файла непросто, поскольку при этом нужно перезаписать оставшуюся часть файла на новое место и удалить образовавшееся свободное место. Предположим, например, что мы хотим удалить запись Мария 555-0912 из исходного файла базы данных. При этом нам нужно сдвинуть к началу файла записи о Павле и Ольге:

  Борис[пробел]555-1212[новая строка]Павел[пробел]555-0012[новая строка]
  Ольга[пробел]555-1190[новая строка]

- При поиске информации в текстовом файле приходится последовательно просматривать файл от начала и до конца. В отличие от DBM-файлов, в которых поиск информации выполнять очень просто, поскольку он связан с хэшем, в текстовых файлах нужно анализировать каждую запись на предмет совпадения с шаблоном. А время выполнения этого процесса зависит от размера текстовой базы данных.

Вставка и удаление записей из текстового файла

Несмотря на перечисленные в предыдущем разделе недостатки, текстовые файлы баз данных все же не так плохи, особенно когда они имеют небольшой размер. Операции вставки и удаления записей из текстовой базы данных выполняются быстро и без особых проблем, если рассматривать текстовый файл как одномерный массив. Например, если база данных имеет вид

```
Борис  555-1212
Мария  555-0912
Павел  555-0012
Ольга  555-1190
```

и сохранена в текстовом файле под именем phone.txt, то написать на Perl короткую программу, загружающую содержимое файла в массив, совсем несложно. Вот один из вариантов программы:

```
#!/usr/bin/perl -w
use strict;

sub readdata {
    open(PH, "phone.txt") ||
       die "Ошибка при открытии phone.txt: $1";
    my(@DATA)=<PH>;
    chomp @DATA;
    close(PH);
    return(ШТА);
}
```

В этом примере функция `readdata()` предназначена для считывания файла phone.txt и помещения его содержимого в массив `@DATA`. При этом из элементов массива удаляются символы конца строки. Если добавить еще одну функцию, `writedata()`, записи базы данных можно будет и читать, и модифицировать. Текст функции `writedata()` выглядит так:

```
sub vritedata {
    my(@DATA)=@_;   # Сохраним в массиве новую запись
    open(PH, ">phone.txt") ||
           die "Ошибка при открытии phone.txt: $1";

    foreach(@DATA) {
        print PH "$_\n";
    }
    close(PH);
}
```

А теперь, чтобы вставить в базу данных новую запись, сначала нужно вызвать функцию readdata(), которая загрузит содержимое файла в указанный массив. После этого для работы с массивом можно использовать функции push, unshift или splice. Завершив все операции с массивом, для сохранения информации в файле вызовите функцию `writedata()`, как показано в следующем примере:

```
@PHONELIST=readdata();     # Поместить все записи базы данных
                           # в массив @PHONELIST
push(@PHONELIST, "Анна 555-1314");
writedata(@PHONELIST);   # Записать массив в файл
```

Чтобы удалить записи из базы данных, примените одну из функций splice, pop или shift к массиву @PHONELIST, а затем запишите содержимое этого массива в файл. Кроме того, содержимое массива можно отредактировать, например с помощью функции grep, выполнив перебор элементов в цикле:

```
@PHONELIST=readdata();   # Поместить все записи базы данных
                         # в массив @PHONELIST
# Удалим все элементы, которые начинаются на "Анн"
@PHONELIST=grep(! /Анн/, @PHONELIST);
writedata(@PHONELIST);
```

В этом примере все записи базы данных сначала копируются с помощью функции readdata() из файла в массив @PHONELIST. Затем с помощью функции grep проверяется, не содержат ли элементы массива строку Анн. Те элементы, в которых такая строка не найдена, снова присваиваются массиву @PHONELIST. И в конце этот массив записывается в файл с помощью функции writedata().

Произвольный доступ к файлу

В предыдущем абзаце, когда мы рассматривали добавление и удаление строк из базы данных, был продемонстрирован один из способов произвольного доступа к файлам. Однако подобный метод нельзя назвать абсолютно безопасным для сохранности данных. Поэтому в следующих разделах мы вкратце рассмотрим несколько средств для чтения и записи файлов с произвольным доступом. Следует отметить, что используются они в программах достаточно редко.

Открытие файлов для чтения и записи

До сих пор мы говорили о трех методах открытия файлов — для чтения, записи и добавления информации в конец файла. Кроме того, файлы можно открывать одновременно и для чтения, и для записи. Возможные режимы открытия файлов перечислены в табл. 15.1.

Таблица 15.1. Возможные режимы открытия файлов

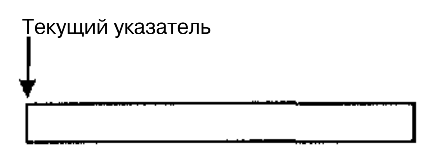
Команда open	Чтение	Запись	Добавление	Создается при необходимости	Старые данные теряются
open(F, "<file") или open(F, "file")	Да	Нет	Нет	Нет	Нет
open(F, ">file")	Нет	Да	Нет	Да	Да
open(F, ">>file")	Нет	Да	Да	Да	Нет
open(F, "+<file")	Да	Да	Нет	Нет	Нет
open(F, "+>file")	Да	Да	Нет	Да	Да
open(F, "+>>file")	Да	Да	Да	Да	Нет

А теперь несколько замечаний.

- По возможности следует избегать использования режимов, при которых информация добавляется в конец файла. В некоторых системах, в частности в UNIX, данные, записываемые в файл, всегда помешаются в его конец, независимо от значения текущего указателя. (О том, что такое указатель, мы поговорим ниже.)
- Никогда не следует использовать режим +>, поскольку при открытии файла его содержимое всегда стирается.

Перемещение по файлу при выполнении операции чтения или записи

При работе с файлом операционная система обычно отслеживает текущее место в нем, из которого должны быть прочитаны или в которое должны быть записаны данные при выполнении следующей операции чтения/записи. Это место называется *указателем* текущей позиции в файле, или просто текущим указателем файла. Например, при открытии файла для чтения текущий указатель файла устанавливается в его начало, как показано ниже.



После того как будет прочитано все содержимое файла, текущий указатель перемещается в конец файла.



Для перемещения текущего указателя в произвольное место файла используется функция seek. Она имеет три аргумента: первый — дескриптор открытого файла, второй — смещение в файле, определяющее новое положение текущего указателя. Последний аргумент определяет относительное положение указанного смещения: 0 — относительно начала файла, 1 — относительно текущей позиции указателя файла, 2 — относительно конца файла. Ниже приведено несколько примеров использования функции seek().

```
# Открываем существующий файл для чтения и записи
open(F, "+<file.txt") || die "Ошибка при открытии file.txt: $!";
seek(F,0,2); # Перемещаемся в конец файла
print F "Это конец файла"; # Добавим текст в конец файла
seek(F,0,0); # Перемещаемся в начало файла
print F "Это начало файла"; # Добавим текст в начало файла
```

Чтобы узнать текущую позицию указателя в файле, используется функция tell. Например, после выполнения фрагмента предыдущего кода функция `tell(F)` вернет значение 16 (длина строки "Это начало файла"). Так происходит потому, что указатель располагается в файле сразу за последней порцией выведенных данных.



В этом разделе мы только поверхностно коснулись функций seek, tell и open, предназначенных для работы с файлами с произвольным доступом. За более подробной информацией обращайтесь к электронной документации. Перечисленные выше функции описаны в разделе perlfunc документации по Perl. Для доступа к нему введите команду perldoc perlfunc. Кроме того, дополнительную информацию по использованию функции open можно найти в разделе perlopentut документации по Perl. Для доступа к нему введите команду perldoc perlopentut.

Блокировка данных

Представьте себе, что вы написали на Perl замечательную программу, которой будут пользоваться многие и многие люди. Независимо от того, в какой операционной системе вы планируете ее эксплуатировать (UNIX, Windows NT или даже Windows 9x), возможны ситуации, когда несколько человек попытаются *одновременно* запустить вашу программу. А если вы предполагаете поместить программу на Web-сервер, она может запускаться так часто, что в памяти вообще одновременно будет находиться несколько копий программы.

А теперь предположим, что вашей программе для работы необходимы данные, хранящиеся, например, в текстовом файле, который был описан выше. Тип файла данных здесь не играет особой роли, поскольку все сказанное ниже можно применить к любому типу баз данных. Рассмотрим приведенный ниже фрагмент кода, в котором используются функции, описанные в предыдущем разделе.

```
chomp($newrecord=<STDIN>);  # Введем запись с клавиатуры
@PHONEL=readdata();         # Загрузим данные в массив @PHONEL
push(@PHONEL, $newrecord);  # Поместим запись в массив
writedata(@PHONEL);         # Запишем массив в файл
```

Вроде бы все выглядит внешне безобидно, не правда ли? Все так, только проблемы начинаются, когда два или несколько человек одновременно запустят вашу программу на выполнение и попытаются добавить в файл новые записи. При этом программа напрочь перестает работать. Ниже приведена диаграмма выполнения программы двумя пользователями, причем второй пользователь начал свою работу сразу после первого. Проанализируйте ее внимательно.

Время	Шаг	Пользователь 1	Пользователь 2
↓	1	`$newrecord="Дмитрий 555-1212";`	
	2	`@PHONEL=readdata();`	`$newrecord="Юрий 555-6611";`
	3	`push(@PHONEL,$newrecord);`	`@PHONEL=readdata();`
	4	`writedata(@PHONEL);`	`push(@PHONEL,$newrecord);`
	5		`writedata(@PHONEL);`

С точки зрения первого пользователя, данные читаются на втором шаге, новая запись "Дмитрий 555-1212" добавляется в массив @PHONEL на третьем шаге, а на четвертом шаге содержимое массива @PHONEL записывается в файл базы данных.

С точки зрения второго пользователя, данные читаются на третьем шаге, новая запись "Юрий 555-6611" добавляется в массив @PHONEL на четвертом шаге, а на пятом шаге содержимое массива @PHONEL записывается в файл базы данных.

Ошибка здесь вот в чем: данные, которые читает второй пользователь на третьем шаге, не содержат записи "Дмитрий 555-1212", поскольку первый пользователь еще не успел ее добавить в файл. Таким образом, второй пользователь добавляет запись "Юрий

555-6611" в массив. @PHONEL, а в это время первый пользователь записывает в файл базы данных содержимое массива @PHONEL, в котором уже есть запись "Дмитрий 555-1212".

Когда же копия программы второго пользователя "добирается" до пятого шага, она "затирает" данные, записанные первым пользователем. Таким образом, в окончательном варианте базы данных на диске будет присутствовать запись "Юрий 555-6611", но не будет записи "Дмитрий 555-1212", что является очевидной ошибкой.



На самом деле проблема гораздо глубже, чем мы только что ее описали. Приведенный выше пример был явно упрощен. Мы не учли тот факт, что функция `writedata()` не может мгновенно открыть файл и записать в него данные. В многозадачных операционных системах ядро обычно прерывает выполнение программы в середине цикла записи и отдает управление другому процессу. По прошествии нескольких десятков миллисекунд (но не мгновенно!) управление снова возвращается программе, которая записывала данные на диск. Таким образом, возможна ситуация, когда обе программы одновременно попытаются записать разные данные в один и тот же файл. Все это может привести к тому, что часть данных будет испорчена или структура файла будет полностью нарушена.

Описанная выше работа программы в многозадачной среде чем-то напоминает гонки "Формулы 1". Отлаживать многозадачные и многопоточные приложения очень сложно, поскольку работа таких программ зависит как от количества запушенных копий, так и от многих других факторов. Поэтому ошибки в многопоточных программах не всегда очевидны и их очень трудно выявить и локализовать.

Как мы уже видели, одновременная запись данных в один и тот же файл несколькими программами — весьма опасная вещь. Однако это совсем не означает, что подобное невозможно. Процесс одновременной записи в файл синхронизируется механизмом, называемым *блокировкой.* Таким образом, блокировка файлов предотвращает одновременную запись в файл со стороны нескольких программ в один и тот же момент времени,

Блокировка файлов вызывает сразу несколько проблем, причем основная из них связана с тем, что в разных файловых и операционных системах используются разные механизмы блокировки. Поэтому в следующих разделах мы расскажем о том, как преодолеть все эти проблемы.

Блокировка в UNIX и Windows NT

ДЛЯ блокировки файлов в системах UNIX и Windows NT используется функция Perl flock, в которой реализован так называемый *совещательный* механизм блокировки. Это означает, что любая программа, которая хочет записать что-либо в файл, должна перед этим вызвать функцию flock и убедиться, что никакая другая программа в данный момент времени ничего не пишет в этот же файл. Естественно, при желании любая программа сможет в любой момент записать данные в файл, не прибегая к средствам блокировки. В этом заключается отличие совещательного механизма блокировки от *принудительного.*

С механизмом совещательной блокировки вы должны быть уже хорошо знакомы. Вспомните, как работает светофор на перекрестке. Красный сигнал светофора запрещает движение транспорта и таким образом препятствует движению машин в пересекающихся направлениях. Но регулировка с помощью светофора работает только тогда, когда все водители строго соблюдают правила дорожного движения. То же самое можно сказать и о механизме блокировки файлов. Любая программа, в которой не исключена возможность доступа к файлам одновременно с другими программами, должна использовать функцию flock для предотвращения нежелательных последст-

вий. Следовательно, механизм совещательной блокировки не препятствует *доступу* нескольких программ к файлу, а предотвращает только возможность получения *права* доступа к файлу.

У функции flock предусмотрены два параметра — дескриптор файла и тип блокировки, как показано ниже.

```
use Fcntl qw(:flock);
flock(дескриптор_файла, тип_блокировки);
```

Функция flock возвращает истинное значение, если блокировка файла была успешно выполнена. В противном случае возвращается ложное значение. Иногда вызов функции flock приостанавливает выполнение программы до момента снятия других блокировок. Ниже мы остановимся на этом более подробно. Директива use Fcntl qw(:flock) позволяет использовать символические имена вместо трудно запоминаемых цифр при определении типа блокировки.

Существуют два вида блокировки: совместно используемая и монопольная. Обычно при чтении файла применяется совместно используемый тип блокировки, а при записи — монопольный. Если какой-либо процесс получил монопольный доступ к файлу, он может выполнять с ним любые действия, поскольку в этот момент никакой другой процесс не сможет обратиться к файлу. Однако многие процессы могут совместно использовать один и тот же файл для чтения, если никакой другой процесс не запросил монопольный доступ к этому файлу.

Ниже мы приведем несколько значений параметра типа блокировки для функции flock.

- LOCK_SH — это значение определяет совместно используемый тип блокировки файла. Если какой-либо другой процесс запросил монопольную блокировку, вызов функции flock с параметром LOCK_SH приведет к приостановке выполнения программы до момента снятия монопольной блокировки. И только после того, как монопольная блокировка будет снята, выполняется совместно используемый тип блокировки.

- LOCK_EX — это значение определяет монопольный тип блокировки файла, открытого для записи. Если какие-либо процессы ранее заблокировали этот файл (для монопольного или совместно используемого доступа), вызов функции flock с параметром LOCK_EX приведет к приостановке выполнения программы до момента снятия всех блокировок.

- LOCK_UN — это значение позволяет отменить блокировку файла. Следует отметить, что используется оно сравнительно редко, поскольку при закрытии файла автоматически все не сохраненные данные записываются на диск и отменяется действие всех блокировок. Учтите, что принудительное снятие блокировки с открытого файла может привести к потере данных или нарушению структуры файла.



Блокировки файла, выполненные с помощью функции flock, автоматически отменяются при закрытии файла либо при завершении работы программы (даже если программа завершилась с ошибкой).

При выполнении блокировки файла, который открыт для чтения и записи, могут возникнуть некоторые нюансы. Проблема заключается в том, что открытие файла и выполнение последующей блокировки — двухступенчатый процесс. Другими словами, для того чтобы заблокировать файл, сначала его нужно открыть. Поэтому, если вы от-

кроете файл с помощью оператора open(FH, ">filename"), а затем сразу же попытаетесь его заблокировать с помошью функции flock, содержимое файла будет затерто перед выполнением блокировки (поскольку при открытии использовался оператор усечения файла >). Таким образом, при открытии файла для записи вы можете случайно модифицировать файл, который был заблокирован другими процессами.

Для решения описанной выше проблемы используется так называемый *семафорный файл* (обычный файл, содержимое которого не имеет особого значения; он используется только для выполнения блокировок).

Перед использованием семафорного файла вы должны определиться с его именем. Для выполнения и снятия блокировок используются уже знакомые нам функции, как показано в листинге 15.3. Этот фрагмент кода нельзя назвать законченной программой, но его вполне можно включить в другой проект как самостоятельную часть.

Листинг 15.3. Универсальные функции для блокировки файлов

```
use Fcntl qw(:flock);
# Для семафора можно выбрать любое имя
my $semaphore_file="/tmp/sample.sem";

# Функция для блокировки (ожидает разблокировки бесконечно долго)
sub get_lock {
   open(SEM, ">$semaphore_file")
       || die "Ошибка при созданин сенафора: $!";
   flock(SEM, LOCK_EX) || die "Ошибка при блокировке: $!";
}

# Функция для снятия блокировки
sub release_lock {
   close(SEM);
}
```

Описанные в листинге 15.3 функции для блокировки и разблокировки можно использовать в любом месте программы, где нужно, чтобы некоторый участок кода выполнялся строго последовательно (без запуска конкурентных процессов), а не только для блокировки файлов при их чтении и записи. Например, даже если приведенный ниже фрагмент кода будет запущен параллельно сразу несколькими процессами, сообщение будет выводиться на экран в текущий момент времени только одним процессом.

```
get_lock();    # Ждем, пока освободится семафор
print "Hello, World!\n";
release_lock(); # Освободим семафор
```

Функции get_lock() и release_lock{), рассмотренные выше, мы будем при необходимости использовать далее по всей книге для выполнения блокировки файлов.



Не рекомендуется устанавливать блокировку перед вводом информации с клавиатуры или для выполнения любой другой медленной операции, поскольку при этом будут приостановлены все другие процессы, ожидающие снятия блокировки. Старайтесь блокировать критичные участки кода только на короткий промежуток времени.

Чтение и запись файлов с блокировкой

А теперь настало время продемонстрировать работу функций **readdata()** и writedataO, выполняющих блокировку текстовой базы данных. Для этого нам нужно определить имя файла-семафора и добавить в код вызовы функций get_lock() и release_lock{}, описанных в предыдущем разделе. Код этих функций мы поместили в начало листинга 15.4.

Листинг 15.4. Пример выполнения операций чтения и записи текстовой базы данных с блокировкой

```
#!/usr/bin/perl -w
use strict;
use Fcntl qw(:flock);

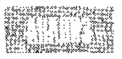
my $semaphore_file="/tmp/list154.sem";

# Функция для блокировки (ожидает разблокировки бесконечно долго)
sub get_lock {
   open(SEM, ">$semaphore_file")
       || die "Ошибка при создании семафора: $1";
   flock(SEM, LOCK_EX) || die "Ошибка при блокировке: $!";
}

# Функция для снятия блокировки
sub release_lock {
    close(SEM);
}

sub readdata {
   open(PH, "phone.txt") ||
        die "Ошибка при открытии файла phone.txt: $!";
    my(@DATA)=<PH>;
    chomp(@DATA);
    close(PH);
    return(@DATA);
}
sub writedata {
     my(@DATA)=@_;
   open(PH, ">phone.txt") \\
       die "Ошибка при открытии файла phone.txt: $1";
     foreach(@DATA) {
             print PH "$_\n";
      }
    close(PH);   # А также снимает блокировку
>
my @PHONEL;

get_lock();
@PHONEL=readdata();
push(@PHONEL, "Николай 555-1012");
writedata( @PHONEL);
release_lock();
```



Большая часть кода из листинга 15.4 вам уже должна быть хорошо знакома. функции get_lock{}, release_lock(), readdata() и writedata() были описаны выше на этом занятии.

Основная часть программы начинается со строки 34. Сначала выполняется блокировка файла с помощью функции get_lock(). Затем с помощью функции readdata() содержимое файла базы данных помещается в массив @PHONEL, выполняется добавление новой записи и содержимое массива записывается обратно в файл базы данных с помощью функции writedata(). После того как будут выполнены все операции, блокировка снимается с помощью функции release_lock{}. В результате другие программы смогут получить доступ к нашей базе данных.

Блокировка в Windows 9x

Как оказалось, в системах Windows 95/98 не поддерживается блокировка файлов. Почему это происходит? Причина заключается в том, что в этих операционных системах только одной программе разрешается открывать файл по записи в данный момент времени, поэтому блокировка оказывается ненужной. В результате, если выполнить функцию flock в системе Windows 9x, будет выдано следующее сообщение об ошибке:

flock() unimplemented on this platform at line...
(Функция flock{}) не реализована на этой платформе...)

К сказанному выше остается добавить, что операционная система Windows 9x является однопользовательской.



В этой книге в листингах примеров, связанных с блокировкой, используются функции get_lock() и release_lock(). Использование этих функций в системах Windows 95 и 98 вызывает появление сообщения об ошибке, поскольку, как уже было сказано выше, в этих операционных системах функция flock не реализована. Поэтому, чтобы примеры работали без ошибок, просто закомментируйте вызовы функций get_lock() и release_lock(). Для напоминания в текст листингов будут включены соответствующие комментарии.

Блокировка в системах UNIX и Windows NT

В некоторых случаях требуется, чтобы несколько программ одновременно могли читать и записывать данные в файл, однако при этом функция flock по каким-либо причинам оказывается недоступной. И даже на тех платформах, где функция flock реализована, бывают ситуации, когда ею нельзя воспользоваться. Например, в системах UNIX эта функция не работает в сетевой файловой системе (NFS). Кроме того, часто программы запускаются в смешанной среде с выделенным сервером на базе UNIX и рабочими станциями на базе Windows. При этом функция flock поддерживается в системах UNIX и не поддерживается в системах Windows.

За более подробной информацией о блокировке файлов и функции flock обратитесь к разделу 5, "Files and Formats", списка часто задаваемых вопросов по Perl. Для этого откройте раздел perlfaq5 электронной документации по Perl.

Резюме

На этом занятии было рассмотрено несколько методов долговременного хранения данных. Сначала мы изучили файлы DBM и способы их привязки к хэшам в программах на Perl, а затем рассмотрели способы использования текстовых файлов для создания простейших баз данных. И в конце занятия была описана проблема одновременного доступа к файлам нескольких программ. Вы узнали, как с помощью блокировок разрешать конфликты доступа к данным и сохранять целостность базы данных.

Вопросы и ответы

Можно ли сохранять структуры данных, описанные на 13-м занятии, "Структуры и ссылки", в DBM-файле или текстом файле?

Если ответить с ходу, то нет, хотя в принципе это возможно, но довольно сложно. Для начала нужно преобразовать "структуру" в строку, которая будет представлять данные и саму структуру, их содержащую. После этого полученную строку нужно использовать как значение ключа в хэше, связанном с DBM-файлом. В Perl предусмотрен специальный модуль, который все это делает автоматически. Его имя — `Data::Dumper`.

Как можно заблокировать **DBM-файл?**

DBM-файлы блокируются с помощью системы семафоров, которая была описана на этом занятии. Вам нужно использовать функции `get_lock()` и `release_lock()`, описанные в листинге 15.3. Поместите эти функции перед открытием DBM-файла и после его закрытия, как показано в следующем примере:

```
get_lock();
dbmopen(%hash, "foo", 0644) [J die "dmbopen: $!";
$hash{newkey}="Значение";
dbmclose(%hash);
release_lock();
```

Можно каким-нибудь образом проверить, приведет ли вызов функции `flock` к паузе в работе программы без реальной приостановки выполнения программы?

Да, это возможно. У функции flock предусмотрено специальное значение параметра, использование которого не приводит к приостановке выполнения программы. Такой вызов функции flock называется *неблокирующим*. Чтобы проверить, вызовет ли функция flock приостановку в выполнении программы, поместите значение `|LOCK_NB` после типа блокировки, как показано ниже.

```
use Fcntl qw(:flock);
# Попытка выполнять монопольный захват без перевода
# программы в состояние ожидания
if (not flock(LF, LOCK_EX|LOCK_NB)) {
    print "Нельзя выполнить блокировку: $!";
}
```

Более того, вы даже можете перевести программу на некоторое время в состояние ожидания, а затем вывести соответствующее сообщение, если в конечном итоге не удастся выполнить блокировку через заданное число попыток. Вот пример:

```
use Fcntl qw(:flock);
$lock_attempts=3;
while {not flock(LF, LOCK_EX|LOCK_NB)) {
   sleep 5; t Ждем 5 секунд
   $lock_attempts~;
   die "Нельзя выполнить блокировку!" if (not $attempts);
}
```

Семинар

Контрольные вопросы

1. В хэшах, связанных с DBM-файлами, ключи могут иметь любую длину.

   а) да;

   б) нет.

2. Почему так сложно вставить данные в середину текстового файла?

   а) потому что при этом нужно освободить место под вставку новых данных и переместить данные файла на новое место;

   б) текстовые файлы нельзя одновременно открыть и для чтения и для записи;

   в) текстовый файл нужно сначала заблокировать, а затем редактировать.

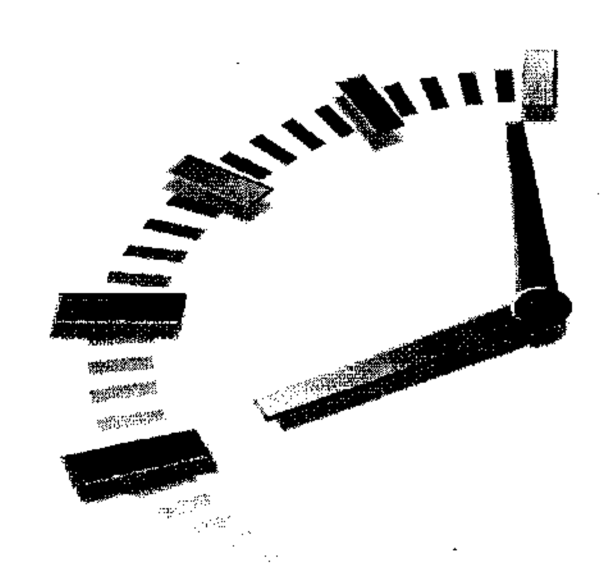
3. В каком разделе списка часто задаваемых вопросов описан процесс блокировки файлов?

Ответы

1. Правильный ответ — вариант б). По умолчанию в DBM-файлах общая длина ключей и связанных с ними данных не должна превышать 1024 символа.
2. Правильный ответ — вариант а). Фрагменты данных в текстовых файлах нельзя взять и переместить вверх или вниз по файлу без соответствующего перемещения на новое место прилегающих к ним данным. Вариант ответа в) также правильный, но только в том случае, если с файлом одновременно работает несколько программ.
3. Раздел 5, "Files and Formats".

Упражнения

- Напишите простую программу, которая увеличивает значение счетчика, хранящегося в файле. Например, сделайте так, чтобы счетчик увеличивался на 1 при каждом запуске программы. Не забудьте использовать средства блокировки, чтобы можно было запускать одновременно несколько копий вашей программы.



16-й час

Сообщество Perl

На этом занятии вы можете немного передохнуть. Итак, усядьтесь поудобнее, запаситесь чем-нибудь вкусненьким и послушайте рассказ об истории и культуре Perl.

Можно было бы ожидать, что подобный материал окажется в приложении или во введении. Но эти разделы в любой книге в лучшем случае бегло просматривают, а мне не хотелось бы, чтобы вы упустили такие важные сведения. Чтобы использовать весь потенциал Perl, вы должны иметь хоть небольшое представление о сообществе Perl.

Зная, чем живет сообщество Perl, вы сможете понять, какие ресурсы имеются в вашем распоряжении, почему они находятся именно там, а не здесь, как они работают и почему Perl таков, каков он есть. Существует множество ресурсов, которые могут вам пригодиться, и эта глава поможет найти их.

Основные темы этого занятия.

- Немного об истории Perl.
- Что такое CPAN и как им пользоваться.
- Куда обратиться за помощью.

Так что же такое это сообщество Perl?

Чтобы получить представление о культуре Perl, о том, как функционирует сообщество Perl и какие ресурсы имеются в наличии, необходимо понять, чем живет сообщество Perl.

Краткая история Perl

В 1988 году Internet была совершенно другой. Во-первых, она была намного меньше, а во-вторых, "выглядела" совсем не так, как сегодня. В то время к Internet было подключено приблизительно 60 000 компьютеров. Сегодня это число превышает 10 миллионов и растет с каждым днем.

В то время Всемирная информационная сеть (WWW) не существовала. Мысль о ней зародилась только в 1991 году в Европейской лаборатории физики заряженных частиц (CERN), а первый графический броузер, Mosaic, был создан лишь в 1993 году.

Большая часть передаваемой по Internet информации была текстовой. Сеть новостей Usenet обеспечивала такую систему передачи сообщений, при которой члены групп по интересам могли связываться между собой и быть в курсе последних новостей в той или иной области. Электронная почта существовала практически в том же виде, в каком она есть сегодня, т.е. главным образом — в текстовом. Передача информации в Internet ограничивалась пересылкой файлов и подключением к удаленным компьютерам с помощью telnet.

В январе 1988 года Лэрри Уолл объявил, что он только что написал программу, заменяющую утилиты awk и sed системы UNIX и назвал ее "Perl". В первом руководстве по Perl дается его описание.

Perl — это интерпретируемый язык, оптимизированный для обработки произвольных текстовых файлов, извлечения информации из этих текстовых файлов и печати отчетов на основе этой информации. Это также хороший язык для выполнения многих системных задач. Perl написан не для красоты, а для дела. Это означает, что во главу угла ставились такие качества, как простота в использовании, эффективность, полнота, а не изящность и требования компактности кода. Perl объединяет в себе (во всяком случае, по мнению автора) некоторые из лучших черт языков C, sed, awk и sh, поэтому у людей, знакомых с этими языками, не должно быть с ним особых трудностей. (Историки языков программирования найдут также некоторые черты csh, Pascal и даже BASIC-PLUS.) Синтаксис выражений Perl очень близок синтаксису выражений языка C. Если у вас есть задача, для решения которой обычно используются средства sed, awk или sh, но их возможностей в данном случае оказывается недостаточно, или выполнение должно идти намного быстрее, и вам кажется, что глупо писать эту программу на C, то, возможно, вам нужен именно Perl. Кроме того, существуют трансляторы, позволяющие преобразовать sed- и awk-сценарии в Perl-сценарии.

Вторая версия Perl была выпущена в июне 1988 года. Она уже была очень похожа на современный Perl: большинство функций Perl 2 можно найти в сегодняшней версии. Это был и есть полностью функциональный язык программирования, обладающий богатыми возможностями. Как сказано в описании, разработка функций Perl в то время была направлена на решение задач обработки текста и системного программирования.

Для Perl 1991 год стал рекордным. В январе было опубликовано первое издание книги *Programming Perl,* авторы Лэрри Уолл и Рендал Шварц (Randal Schwartz). Эта книга была (и остается, судя по последним изданиям) полным справочником по языку Perl. На розовой обложке был изображен верблюд — официальный талисман языка Perl. (Это животное не слишком красивое, но верное, надежное и очень полезное.)

Данная публикация совпала по времени с выходом Perl 4. Эта версия была первой широко распространяемой версией Perl, и ее следы еще можно найти сегодня в разных уголках необъятной Сети, несмотря на то что последние исправления вносились в нее в 1992 году. Если она вам случайно попадется, не стоит ею пользоваться.

В октябре 1994 года была выпущена пятая версия Perl. В нее были включены такие возможности, как приватные переменные, ссылки, модули и объекты (с которыми вы еще не знакомы). В октябре 1996 года вышло второе издание книги *Programming Perl* ("The Blue Camel" — "Голубой верблюд". — *Прим. перев.)* с описанием этих новых возможностей.

Открытый код

Одна из причин успеха Perl кроется в принципах его разработки и распространения. Интерпретатор Perl — это часть программного обеспечения, построенного по принципу открытого кода (open source). *Открытый код* — это новый термин, присвоенный программистами старому понятию, а именно бесплатно распространяемому

программному обеспечению. Такую программу можно получить бесплатно, причем любой, кто хочет внести в нее какие-то изменения, может просмотреть, исправить и переделать ее исходный текст. Другими примерами пакетов программного обеспечения, следующих данной модели, являются операционные системы Linux и FreeBSD, Web-сервер Apache и броузер Mozilla.

Использование модели открытого кода — на самом деле очень эффективный способ разработки программного обеспечения. Поскольку код пишется добровольцами, ненужные программы обычно в пакет не включаются, а функции, которые кажутся полезными, предлагаются для включения и включаются в пакет (если они действительно нужны). Качество такого профаммного обеспечения получается довольно высоким, так как каждый, кто интересуется пакетом, имеет право и обязанность внимательно следить за его разработкой и участвовать в поиске ошибок. Чем больше людей будут просматривать исходный код, тем меньше у ошибок шансов выжить.



Эрик С. Реймонд (Eric S. Raymond) написал ряд великолепных очерков о модели разработки программного обеспечения с открытым кодом. Он объяснил, как пришли к этой модели и почему она столь эффективна и экономически выгодна. В первом очерке, "The Cathedral and the Bazaar" ("Храм и Торговля" — *Прим. перев.*), предоставлена хорошая вводная информация о том, как работает модель разработки программ с открытым кодом. URL этих статей вы найдете в разделе "Резюме" в конце данной главы.

Авторское право на интерпретатор Perl принадлежит Лэрри Уоллу; он его владелец и может делать с ним все, что захочет. Но, как и для большинства профамм, для Perl может быть выдано разрешение (лицензия) на его использование. В лицензии на профаммное обеспечение описывается, как его можно использовать и распространять; это та самая информация, набранная мелким шрифтом, которая появляется первой, когда вы начинаете устанавливать купленный в магазине профаммный продукт. Лэрри Уолл предлагает вам на выбор два различных варианта лицензии: GNU General Public License и Perl Artistic License. Прочитав оба варианта, вы можете выбрать один из них и следовать этим условиям соглашения при последующем распространении Perl.

Тексты обеих лицензий довольно объемны, поэтому я вкратце приведу их основные положения.

- Вы можете распространять исходный текст интерпретатора Perl, но должны продублировать соглашение об авторском праве.
- Вы можете модифицировать оригинал исходного кода, но должны четко пометить внесенные изменения как свои собственные и либо отказаться от изменений, либо ясно указать, что это не стандартная версия Perl. Вы обязаны также предоставить стандартную версию Perl.
- Вы можете брать разумную плату за распространение Perl. Вы можете также брать плату за осуществление технической поддержки, но сам Perl продавать нельзя. Можно включать Perl в другие продукты, которые вы продаете.
- Профаммы, написанные на Perl, не подпадают под действие этой лицензии.
- Гарантийных обязательств для Perl нет.

При решении юридических вопросов не следует полагаться только на это резюме; я привел его здесь только с целью дать вам почувствовать, что представляют собой сами лицензии.



Прежде чем пытаться включить Perl в другой программный пакет, очень важно самостоятельно прочитать тексты лицензий и определить, соответствуют ли ваши действия условиям, изложенным в них. Лицензия Perl Artistic License включается в каждую поставку Perl в виде файла с именем `Artistic`. А лицензию GNU General Public License можно просмотреть на Web-сервере по адресу `http://www.gnu.org`.

Данные лицензии позволяют разрабатывать и совершенствовать Perl при открытом обсуждении. Таким образом, весь исходный текст Perl доступен для каждого, кто хочет ознакомиться с ним и предложить внести какие-либо изменения. Подобный подход поощряет качественное программирование и не дает увязнуть в трясине собственнических настроений, скрытно разрабатывая код так, чтобы это было непонятно всем остальным.

Разработка Perl

Разработка интерпретатора, языка и модулей, входящих в стандартную поставку Perl, проходит через список рассылки, где разработчики Perl предлагают изменения, изучают сообщения об ошибках и обсуждают, какие изменения следует внести в исходный текст Perl.

Каждый может участвовать в этом процессе — именно в этом и состоит принцип модели открытого кода. Но, чтобы не допустить хаоса, предлагаемые изменения тщательно изучаются и "фильтруются" группой ведущих разработчиков, которые одобряют или отвергают эти изменения и поддерживают основную линию разработки Perl. Изменения оцениваются исходя из того, что хорошо для Perl, а что — нет, насколько полезны эти изменения, и может ли любой человек нормально их воспринять. Лэрри Уолл, осуществляющий надзор над этим процессом, играет роль доброжелательного диктатора, разрешая вносить полезные изменения и налагая запрет на изменения, которые он считает пагубными.

Выпускаемые версии Perl нумеруются двумя различными способами. До августа 1999 года они нумеровались в формате `major.minor_patchlevel`. Так, `4.036_18` — это четвертая версия Perl, выпуск 36 с "заплатами" уровня 18. Иногда в номерах версий уровень "заплат" не указывается. Текущей версией Perl на момент подготовки к изданию данной книги летом 2000 года была 5.6.

Схема нумерации новых версий более традиционна и имеет формат `major.minor`. Предположительно, следующая за этой версия Perl будет называться 5.7 и т.д.

Сеть полного архива Perl (CPAN)

С целью еще большего расширения среды разработки в Perl предусмотрены дополнительные модули, которые содержатся в CPAN (Comprehensive Perl Archive Network).

Что это такое?

Сеть полного архива Perl (*Comprehensive Perl Archive Network — CPAN)* — это большая коллекция программного обеспечения и документации к Perl. Это программное обеспечение — плод совместных усилий добровольцев, которые захотели внести свой вклад в работу сообщества Perl и написали различные модули, программы и документацию.

Список модулей, имеющихся в CPAN, очень обширный. На момент написания этой книги сеть CPAN существовала приблизительно четыре года и в ней имелось

свыше 3500 готовых к инсталляции модулей. Эти модули охватывают широкий диапазон программистских задач. В табл. 16.2 приведен краткий список данных модулей, который даст вам представление о том, что имеется в CPAN.

Нужно иметь в виду самое важное — для большинства задач уже есть модули, позволяющие решить их хотя бы частично. Эти решения, имеющиеся в CPAN, были запрограммированы и протестированы; многие программисты проверяли эти коды и оценивали степень их полноты и корректности.

Таблица 16.2. Модули, имеющиеся в CPAN

Модуль	Описание
Tk	Графический интерфейс для Perl-программ. Имеются специальные модули инструментальных средств доступа к специализированным графическим библиотекам, таким как Win32 API, Gtk, Gnome, Qt, или набору инструментальных средств X11
Net::*	Сетевые модули. Интерфейсы к службам Mail, Telnet, IRC, LDAP и еще более 40 других
Math::*	Свыше 30 модулей для таких конструкций, как комплексные числа, быстрые преобразования Фурье, операции с матрицами и т.д.
Date::*, Time::*	Модули для преобразования дат/времени в различные форматы и выполнения операций над ними
Data::*, Tree::*	Модули для выполнения операций над структурами данных, такими как связанные списки и двоичные деревья
DBI::*	Общий интерфейс к базам данных
DBD::*	Интерфейс к коммерческим и бесплатным базам данных, таким как Oracle, Informix, Ingres, ODBC, Msql, MySQL Sybase и многим другим
Term::*	Тонкая настройка текстовых терминальных окон, таких как окно сеанса MS-DOS в Windows или Х-терминала в UNIX
String::*, Text::*	Десятки модулей для синтаксического анализа и форматирования текста
CGI::*, URL::*, HTML::*, LWP::*	Модули для создания, обслуживания, извлечения и анализа Web-страниц
GD, Graphics::*, Image::*	Модули для работы с графикой и изображениями в различных форматах
Win32::*, Win32API::*	Модули для работы с системой Microsoft Windows

На все модули, имеющиеся в CPAN, распространяются авторские права их авторов. Поэтому следует прочитать файл README, который есть в каждом модуле, чтобы узнать, на каких условиях можно использовать данный модуль. Чаще всего эти модули распространяются на таких же условиях, как и сам Perl, по лицензии Artistic License или GNU General Public License.

CPAN — это также имя стандартного модуля, который используется как вспомогательный при инсталляции дополнительных модулей в имеющуюся версию Perl. Этот модуль CPAN описан в приложении к этой книге "Инсталляция модулей в Perl".

Почему люди вносят свой вклад в работу сообщества Perl?

За последние полвека, когда родилась и начала бурно развиваться компьютерная индустрия, программисты снова и снова решали одни и те же задачи. Поиск, сортировка, передача информации, чтение, запись — эти задачи мало чем изменились с 50-х годов XX столетия. Некоторые книги по теории компьютерного программирования и управления даже спустя 20 или 30 лет по-прежнему актуальны.

Решение одних и тех же проблем снова и снова не всегда интересно и часто приводит к менее удачным решениям; это называется "изобретать велосипед". И, в конце концов, программистами движет стремление решать интересные задачи.

Очень типичной является ситуация, когда программист тратит много времени и сил на решение сложной задачи только для того, чтобы впоследствии обнаружить, что давно существует гораздо более простое и изящное решение. Пережив подобное разочарование, программисты начинают искать способы обмена программами друг с другом. Система совместного использования программ имеет интересный побочный эффект — качество программ повышается, поскольку другие программисты могут найти в вашей программе недостатки, которые вы не заметили.

CPAN — это попытка сообщества Perl избежать ненужной работы. Модули, находящиеся здесь, должны уберечь вас от разочарования, неизбежно возникающего при повторном изобретении чьего-то "велосипеда".

Качество большинства этих модулей также очень высокое, потому что они, как и Perl, разрабатывались на условиях модели открытого кода. Когда вы устанавливаете модуль на своем компьютере, то автоматически получаете его исходный код. Вы сами можете изучить его и — на основе лицензии — использовать фрагменты исходного кода в своих программах, модифицировать исходный код и даже связаться с автором, если захотите предложить ему внести какие-либо изменения.

Внешне может показаться, что от идеи CPAN веет духом общины. Но истинные причины, по которым авторы вносят свой вклад в CPAN, очень разные. Иногда они поступают так, чтобы сделать доброе дело. Иногда хотят заслужить уважение и восхищение других людей — и надо сказать, что это мощная движущая сила. Но какой бы ни была причина, конечным результатом является огромный объем проделанной работы, т.е. множество модулей, которые вы можете использовать в собственных программах.

Куда двигаться дальше

После прочтения двух третей этой книги вы уже должны разбираться в основах Perl. Разумеется, вы не выучили весь язык целиком. На моей книжной полке по меньшей мере шесть книг по языку Perl, т.е. примерно 2300 страниц, на которых описывается Perl (не принимая в расчет повторяющиеся темы), и все равно некоторые темы сюда не включены.

Вы не найдете единственный ресурс или источник информации, содержащий *все* сведения о Perl. Но в следующих разделах содержатся рекомендации о том, какие шаги следует сделать.

Ресурсы, о которых здесь говорится, представлены примерно в том порядке, в котором их следует разыскивать. Разумеется, и из этого правила есть исключения, но в основном следование указанному порядку поможет вам решить все проблемы настолько быстро, насколько это возможно.

Ваш первый шаг

Если у вас возникла какая-то проблема с Perl, попытайтесь выяснить, какую первую вещь вам трудно сделать. Вы разочарованы и если уже поработали над задачей некоторое время, вероятно, расстроены. Глубоко вздохните, не паникуйте и скажите себе, что все будет хорошо. Хотите верьте, хотите — нет, но это очень важный первый шаг. Большинство людей через некоторое время зацикливаются на задаче и, будучи разочарованы, не могут ясно мыслить. В результате все заканчивается тем, что положение дел только ухудшается.

Прогуляйтесь, выпейте пива, успокойтесь и расслабьтесь. Вы непременно решите эту проблему.

Ваш самый полезный инструмент

Самый полезный инструмент в вашем наборе инструментов Perl — это сам Perl. Прежде всего вам нужно определить, с какого рода проблемой вы имеете дело. Как правило, все проблемы делятся на две категории: синтаксические или логические ошибки.

Если проблема заключается в синтаксисе, то обычно ее, в свою очередь, можно отнести к двум категориям: либо вы что-то сделали неправильно, либо это просто опечатка. Запустите программу и внимательно изучите сообщение об ошибке; обычно Perl правильно определяет, в какой строке что-то не так. А теперь исследуйте эту строки, находя ответы на следующие вопросы.

- В выданном Perl сообщении об ошибке указывается конкретно, где именно ее следует искать? Если да, то ищите ее там! Интерпретатор Perl — ваш самый надежный помощник в деле нахождения ошибок.
- Проверьте, у всех ли скобок (круглых, квадратных и фигурных) есть пары?
- Тщательно ли вы проверили синтаксис операторов? Проверьте его еще раз. Вы будете удивлены, выяснив, как много ошибок на поверку оказываются синтаксическими.
- Может, вы что-то пропустили? Например, точку или запятую?
- Все ли в порядке со строками, находящимися непосредственно перед указанной строкой?
- Если вы вернетесь к разделу данной книги, где говорится об определенном правиле синтаксиса, то сможете ли найти примеры, аналогичные вашему?
- Если вы откуда-то скопировали код, то можете ли поискать этот же фрагмент в другом месте? Возможно, в скопированном коде содержалась ошибка. От них, в конце концов, никто не застрахован.

Если ваша написанная на Perl программа работает, но просто дает неправильные результаты, то, вероятно, есть проблемы с логикой. Но прежде чем разносить все в пух и прах, выполните следующие действия.

1. Убедитесь в том, что в строке `#!` вашей программы содержится ключ `-w`.
2. Убедитесь, что где-то в начале программы есть оператор `use strict`.

Часто оказывается, что многие проблемы, которые внешне кажутся связанными с логикой, — это простые ошибки, легко обнаруживаемые с помощью ключа `-w` и директивы `use strict`. Воспользуйтесь этими средствами и, если проблемы останутся, продолжайте читать дальше.

Отладка программы

ЕСЛИ ВЫ уверены, что синтаксис программы правильный, но просто она дает неверные результаты, то самое время выполнить элементарную отладку.

Первый и, вероятно, самый распространенный метод отладки программы — это использование скромного и непритязательного оператора print. Если им правильно воспользоваться, он может дать некоторую диагностику программы во время ее выполнения, т.е. выдать сообщения о том, что происходит. Посмотрите, как оператор print работает в следующем примере:

```
sub foo {
  my($al, $a2)=@_;
  # Добавлен оператор тестовой печати для целей диагностики
  print STDERR "ОТЛАДКА: Вызов foo с параметрами $al и $a2";
}
```

Но помните, что после завершения работы над программой вы должны убрать из нее все отладочные операторы print. Чтобы впоследствии вам легче было их все разыскать, я рекомендую ввести в них какое-то отличительное ключевое слово, например DEBUG или ОТЛАДКА Кроме того, направляя вывод в стандартный поток ошибок STDERR, вы сможете отделить обычные выходные данные от диагностики. Если вы включили в диагностические сообщения литералы__LIKE__и__FILE__, то Perl напечатает имя текущей строки и файла.

Другой подход, который стоит опробовать, — это отладчик Perl. Этот отладчик можно использовать практически для любой программы, написанной на Perl. Наблюдение за пошаговым выполнением программы может быть очень информативным. Инструкции по использованию отладчика Perl даны на 12-м занятии, "Работа с командной строкой Peri".

Во-первых, помогите себе сами

Если синтаксис вашей программы абсолютно правильный и с логикой, похоже, тоже все в порядке, но результаты получаются не такими, как нужно, то, возможно, пришло время попросить помощи со стороны. За ответами на свои вопросы нужно прежде всего обращаться к документации по Perl.

Как указывалось на 1-м занятии , "Начало работы с Perl", в каждую поставку Perl входит полный набор документации. В поставку выпуска 5.6 включено свыше 1700 страниц документации. В ней описывается каждый модуль и каждая функция языка Perl, т.е. освещаются практически все аспекты; здесь же вы найдете большой список часто задаваемых вопросов (Frequently Asked Questions — FAQ).

Чтобы получить список имеющейся документации, наберите после приглашения команду **perldoc** perl. В ответ будут перечислены все разделы руководства, а также дано общее описание Perl.

В разделе FAQ содержится список вопросов о языке Peri, наиболее часто задаваемых начинающими программистами и профессионалами. Стоит просмотреть данный список хотя бы один раз, чтобы получить представление о том, какого рода вопросы здесь содержатся, даже если вы пока не совсем понимаете смысл ответов.

Если по какой-то причине на вашем компьютере не установлена документация по Perl или в ответ на команду perldoc она не появляется, то прежде всего вам следует поговорить с системным администратором и попытаться найти документацию. Очень важно, чтобы документация была установлена правильно, потому что электронная документация соответствует версии Perl, которую вы используете. Любая другая документация, скорее всего, будет иметь какие-то отличия.

Если вы не можете получить доступ к электронной документации, то сможете найти ее также на Web-сервере по адресу `http://www.perl.com`.

Учитесь на ошибках других

Usenet — это система пересылки сообщений, которая была разработана в начале 80-х годов и постепенно распространилась по Internet, которая еще только набирала силу. Usenet — это десятки тысяч дискуссионных групп, посвященных самым разным темам, начиная от медитации, садоводства, компьютеров и научной фантастики и заканчивая хоккеем и роликами. Наряду с этим существуют также местные группы новостей для каждого региона в мире. А вот список групп новостей, посвященных Perl.

`comp.lang.perl.announce`	Новости о новых выпусках и модулях Perl, а также другая информация
`comp.lang.perl.moderated`	Группа с небольшим трафиком, где обсуждение проблем, связанных с Perl, ведется строго по правилам, за которыми следит модератор
`comp.lang.perl.misc`	Дискуссионная группа с высоким трафиком, где обсуждается все, что связано с Perl

Для того чтобы получать новости Usenet, вам нужна специальная программа чтения новостей. Такую программу найти несложно. Можно зайти на любой сервер, откуда загружают программное обеспечение, и взять программу чтения новостей. Можно зайти также на некоторые Web-серверы (например, deja.com или supernews.com), которые являются зеркальными отражениями групп новостей Usenet в формате Web; здесь для чтения новостей вам потребуется только Web-броузер.

В этих группах новостей люди задают вопросы о проблемах, которые у них возникли с Perl, а другие люди отвечают на эти вопросы — причем все это делается на добровольной основе. Кроме того, здесь проводятся обсуждения тем, касающихся Perl, которые имеют всеобщий интерес.

Однажды я сделал наблюдение, правильность которого неоднократно подтверждалась на протяжении всей моей карьеры программиста: "В программировании нет новых проблем". Можете быть совершенно уверенны в том, что с любой возникшей у вас проблемой кто-то уже сталкивался. Весь фокус в том, чтобы найти задававшего этот вопрос и узнать, какой ответ он получил. Весьма вероятно, что по крайней мере один человек задавал вопрос, очень похожий на ваш, в одной из этих групп новостей.

На сервере deja.com ведется история большей части Usenet. Если вы воспользуетесь поисковой системой этого сервера, введя несколько точно подобранных ключевых слов, то, скорее всего, найдете ответы на свой вопрос.

Рассмотрим такой пример. Предположим, вам нужно узнать, как написать программу на Perl, чтобы "вытащить" Web-страницу. Перейдите к странице Power Screen сервера deja.com и заполните пустые поля в этом окне следующей информацией:

Keywords: fetch web page
Forum: `comp.lang.perl.misc`

Для данного примера все остальные поля оставьте пустыми. Когда будут выданы результаты поиска — примерно 100 вариантов соответствий, — то большинство из них будут относиться к теме, о которой вы спрашивали. По поводу статей, которые вы будете читать в Usenet, нужно иметь в виду следующее.

- Не все ответы правильны. Любой может задать вопрос и любой может на него ответить. Прочтите несколько ответов и решите для себя, какие из них заслуживают доверия. При этом польза, которую вы извлечете, может быть разной.

- Если вы не уверены в правильности ответа, используйте его в качестве отправной точки и проверьте предоставленную информацию самостоятельно. Теперь, когда вы знаете, где искать, прочитайте соответствующие страницы руководства, посвященные данной теме.

- На сервере deja.com сохраняется архив новостей за последние пять лет. Ответы, которые были правильны пять лет назад, теперь могут быть опротестованы.

Когда все остальное не удалось, спрашивайте

ЕСЛИ ВЫ просмотрели электронную документацию, книги, архивы Usenet и так и не нашли ответа на свой вопрос, то нужно кого-то спросить.

Просьба о помощи — это последнее средство, но никак не первое. Уникальным способом получения ответов на вопросы является обращение к специалистам. Только они в ответ на плохо сформулированный вопрос могут иногда выдавать гениальные решения проблем. Но, в отличие от всех остальных ресурсов, о которых я упоминал, возможности людей отвечать на вопросы являются ограниченными. Они устают, возможно, у них выдался тяжелый день, но особенно они могут уставать оттого, что им приходится снова и снова отвечать на одни и те же вопросы.

И хотя вполне вероятно, что человек, которого вы спрашиваете, знает ответ, помните, что вы отнимаете у него время и заимствуете его опыт. Поэтому, прежде чем беспокоить кого-то своими вопросами, вы обязаны серьезно потрудиться и провести самостоятельный поиск.

Чтобы задать вопрос в Usenet, воспользуйтесь программой чтения новостей или одним из упомянутых выше Web-интерфейсов. Формулируя вопрос, придерживайтесь следующих правил.

1. Прежде чем что-либо сделать, выясните, есть ли у группы новостей список часто задаваемых вопросов. У групп новостей, посвященных Perl, такой список есть. Кроме того, он входит в поставку интерпретатора Perl. Если же вас интересует любая другая группа новостей, поищите FAQ этой группы на сервере `deja.com`, прежде чем посылать в группу свое сообщение.

2. Задавайте вопрос там, где нужно, т.е. в соответствующей группе новостей. Общий вопрос о языке Perl нужно задавать в группе `comp.lang.perl.misc`. А вопрос по программированию, касающийся CGI, видимо, следует задавать в `comp.infosystems.www.authoring.cgi`. Прочитав FAQ группы, вы поймете, в нужном ли месте задаете вопрос.

3. **Выберите** для своего сообщения хорошее название темы в строке Subject. Оно должно точно описывать проблему. Старайтесь избегать лишних слов (типа "помогите" или "вопрос новичка"); пусть название темы будет содержательным, но лаконичным.

4. В теле сообщения обязательно должно содержаться следующее:

   а) описание того, что вы пытаетесь сделать (и, возможно, даже описание того, зачем это нужно);

   б) описание того, что вы пытались делать до сих пор и что получили в итоге;

   в) описание ошибок, с которыми вы столкнулись.

   Если вы публикуете в группе новостей сообщения об ошибках или ссылки на свою программу, обязательно включите достаточный фрагмент кода,

чтобы ваши респонденты могли понять, что происходит. Если вы пытаетесь обработать данные, поместите несколько строк для примера.

В теле сообщения не должно содержаться следующее:

а) большие фрагменты кода;

б) двоичные файлы, такие как .EXE-файлы, закодированные утилитой uuencode;

в) вложения MIME. Вместо этого включите в тело сообщения свои примеры и код.

5. Правильно указывайте адрес электронной почты — на случай, если кто-либо захочет ответить вам, но не публично, а конфиденциально.

6. И, самое главное, будьте вежливы. Вы просите об одолжении незнакомых людей. Причем никто не обязан помогать вам. Говорите "спасибо" и "пожалуйста" и избегайте замечаний, которые могут вызвать чье-то недовольство или возмущение. Пытаясь получить помощь, не пользуйтесь хитростями и уловками. Например, не пишите "Помогите бедной девочке с ее CGI-программой..." или "Я предоставлю вам бесплатную Web-страницу, если вы...". Подобные уловки примитивны и унизительны.

Если вы опубликовали свое сообщение в группе новостей, спокойно ждите ответа. Пройдет несколько дней, прежде чем новости Usenet распространятся по всему миру. Кроме того, люди не всегда следят за новостями и читают каждое сообщение. Будьте терпеливы и, пока ожидаете, займитесь другой задачей. И, что[1] бы вы ни делали, не посылайте снова в Usenet свой вопрос слишком быстро. Подождите хотя бы пару недель, прежде чем снова задать свой вопрос. Причем сначала перефразируйте его, проверьте, четко ли сформулирована тема в строке Subject, а затем сделайте еще одну попытку получить ответ.

Ответы на ваш запрос могут начать поступать немедленно (в течение нескольких минут) либо появятся спустя месяц или более со дня опубликования. Как я уже говорил, качество ответов будет меняться в широком диапазоне. Одни ответы будут информативными, в то время как другие могут оказаться просто неправильными. Одни респонденты будут благожелательными и вежливыми, а другие — ужасно грубыми. По правилам этикета Сети следует поблагодарить всех ответивших и проигнорировать любые неприятные, оскорбительные или возмущенные письма, которые вы получили.

Другие ресурсы

Если вы хотите больше узнать о Perl, о программировании на Perl и о сообществе Perl, обратитесь к следующим дополнительным ресурсам.

- Книга Larry Wall, Tom Christiansen, Randal Schwartz, *Programming Perl* считается у программистов на Perl настольной. После того как вы изучите основы Perl, можете использовать ее в качестве отличного справочника.

- Книга Tom Christiansen, Nathan Torkington, *Perl Cookbook*, написанная в стиле детального справочного руководства, содержит обширную коллекцию различных задач, примеров, решений и комментариев к сотням задач. Сначала формулируется задача, потом описывается ее решение, а затем приводятся примеры и объяснения этого решения.

- *The Perl Journal*. Этот журнал, выходящий раз в три месяца, рекомендует себя как "Голос Сообщества Perl". Это настоящий технический журнал, авторами статей которого являются члены Сообщества Perl (программисты, использую-

шие Perl ежедневно), а не ученые мужи или профессиональные писатели. Вот цитата из первого номера этого журнала: "Мы стремимся к тому [...], чтобы быть интеллектуальным изданием, стимулирующим изучение языка Perl, ремесла программирования, а также некоторых других ремесел..."

Дополнительную информацию по этим вопросам можно найти в следующих источниках.

- Internet History (История Internet): Hobbe's Internet Timeline

  `http://www.isoc.org/zakon/Internet/History/HIT.html`

- History of Perl (История Perl): CPAST

  `http://history.perl.com`

- The Perl Journal

  `http://www.tpj.com`

- C PAN

  http://www.perl.com/CPAK

- Электронная документация

  Должна бытъ установлена на вашем компьютере. См. также `http://www.perl.com`

- Очерки Эрика С. Реймонда о модели открытого кода

  `http://www.netaxs.com/~esr/writings`

Резюме

На этом занятии вы узнали немного об истории Perl и использовании для Perl модели разработки открытого кода. Вы выяснили также, что такое CPAN, зачем он существует и кто его поддерживает. Наконец, вы узнали, какие ресурсы помогут вам, если у вас возникнут какие-либо проблемы с написанием программ на Perl.

Вопросы и ответы

Если Web была изобретена после Perl, то почему он является языком написания CGI-программ?

Perl является языком написания CGI-программ по той же причине, по которой компьютеры используются для игр (хотя они были изобретены не для этого), — они просто хорошо подходят для этой цели. На следующем занятии подробно объясняется, почему Perl — хороший язык для написания CGI-программ.

Я опубликовал сообщение в Usenet и получил грубый, раздраженный ответ. Что мне делать?

Для начала выясните, нет ли в этом письме какого-нибудь хорошего совета? Если есть, примите его, а на грубость не обращайте внимания. Если нет — проигнорируйте это письмо. Жизнь слишком коротка, чтобы тратить ее на "эпистолярную войну".

Существует ли простой способ поиска в CPAN?

Да! На Web-странице `http://search.cpan.org` есть общая функция поиска. Кроме того, можно просмотреть последние изменения и провести поиск модулей по категориям.

Семинар

Контрольные вопросы

1. В какую группу новостей Usenet нужно сначала посылать вопросы по созданию CGI-приложений на Perl?

   а) `comp.infosystems.www.authoring.cgi`;

   б) `comp.lang.perl.misc`.

2. Если на вашем компьютере, похоже, не установлена система электронной документации, что нужно делать?

   а) попросить администратора установить ее;

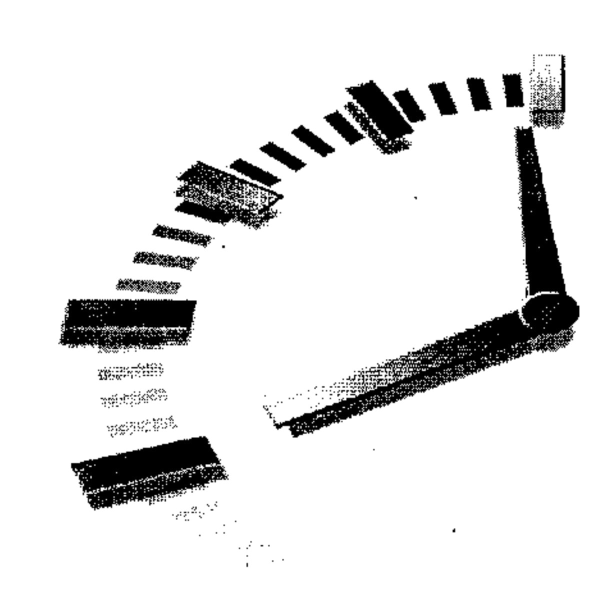
   б) послать сообщение в `comp.lang.perl.misc`;

   в) попробовать найти документацию в другом месте, например на сервере `http://www.perl.com`.

Ответы

1. Правильный ответ— вариант а). Группа новостей `comp.lang.perl.misc` может быть вторым местом, куда следует посылать вопросы, касающиеся создания CGI-программ, но отнюдь не первым.

2. Правильным будет и вариант а) и в), и, видимо, именно в этом порядке. Возможно, документация уже установлена на вашем компьютере и администратор поможет найти ее. В противном случае на сервере www.perl.com есть свежий набор документации.



Часть III

CGI-программирование на Perl

Темы занятий



17-й час

Введение в CGI

Небывалый рост популярности Internet — это, несомненно, заслуга World Wide Web. После появления первого графического Web-броузера в 1993 году сеть Internet развивалась семимильными шагами. Тогда количество узлов Internet удваиваюсь каждые 20 месяцев, а теперь удвоение количества узлов происходит еще быстрее — каждые 12 месяцев. Количество внутренних сетей, так называемых intranet, увеличивается еще стремительнее.

Содержимое Web заметно усложнилось с 1993 года. Теперь пользователи ожидают от Web нечто большее, чем просто показ статических Web-страниц. Популярные Web-серверы содержат динамические Web-страницы с непрерывно обновляемой информацией. Поддержка сложных страниц с непрерывно изменяющимся содержимым без использования CGI практически невозможна. CGI — это сокращение от *Common Gateway Interface (Интерфейс общего шлюза).*



Для полноценного усвоения материала следующих семи занятий вам потребуются определенные знания языка гипертекстовой разметки HTML. Если вы не знакомы с HTML — не отчаивайтесь. Изучить его довольно легко, особенно в объемах, необходимых для данной книги.

HTML используется при создании Web-страниц. Простой текст в формате HTML содержит специальные коды форматирования, содержащие информацию о том, каким образом выделенные элементы текста должны отображаться в Web-броузере. Например, фрагмент HTML `<i>не</i>` сложен в изучении — вполне обычный текст за исключением маркеров <ix/i>. Эти маркеры называются *дескрипторами* и служат для обозначения необходимого форматирования при отображении текста. В данном случае слово не должно по возможности (поскольку не все броузеры графические) отображаться в Web-броузере курсивом.

Более подробное описание HTML выходит за рамки данной книги. Дело не в сложности этого материала, а в его обилии. Спецификация HTML разрабатывается консорциумом World Wide Web (W3C), ее адрес в Internet — http://www.w3c.org. На этом же Web-сервере вы найдете несколько превосходных учебников по HTML. Для изучения HTML можно также порекомендовать книгу *Использование HTML 4. Специальное издание,* выпущенную издательским домом "Вильямс".

Основные темы этого занятия.

- Как работает Web.
- Что нужно знать *перед* тем, как создавать CGI-программы.
- Создание первой CGI-программы.

Просмотр содержимого Web

Работа Web заключается в организации взаимодействия двух различных систем, обменивающихся данными. Система, которая загружает Web-страницу, называется *клиентом.* Обычно для этого на клиентской системе запускается программа *Web-броузер,* такая как Netscape Communicator, Internet Explorer, Opera или какая-либо другая. Так вы получаете доступ в пространство World Wide Web. Web-броузер обеспечивает вас средствами навигации в этом пространстве и отображения Web-страниц.

На другом конце Web-соединения находится система, называемая *Web-сервером.* Она принимает клиентский запрос на определенную страницу, находит ее на локальном диске и посылает клиенту — Web-броузеру. Схема, иллюстрирующая подобное взаимодействие, приведена на рис. 17.1.

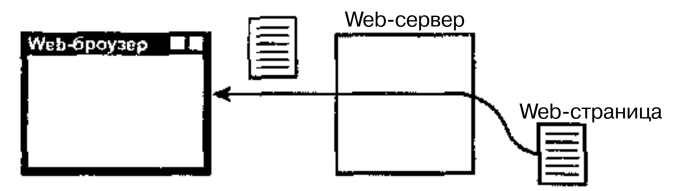


Рис. 17.1. Процесс загрузки Web-страницы Web-броузером

Загрузка статической Web-страницы

Клиент запрашивает Web-страницу по ее URL (Uniform Resource Locator — унифицированный локатор ресурсов), содержащему информацию об адресе сервера и типе поддерживаемого сервером протокола. Обычный URL выглядит следующим образом:

```
http://www.google.com:80/more.html
```

Этот URL состоит из нескольких частей.

- http — обозначение протокола. Протокол пересылки гипертекста HTTP служит для пересылки Web-страниц. В этом месте URL можно столкнуться с протоколом пересылки файлов ftp или так называемым безопасным HTTP (https, или secure HTTP).
- `www.google.com` — адрес сервера, также называемый *именем компьютера.* На этом сервере находится интересующий вас документ. Кроме имени компьютера в этом месте может быть его IP-адрес, который выглядит как четыре числа, разделенных точками: 209.185.108.147. Для доступа к World Wide Web IP-адреса используются сравнительно редко, поскольку они менее надежны, чем имена.
- :80 — номер порта, по которому будет осуществляться соединение клиента и сервера. Эта часть URL необязательна. Номер порта, как правило, определяется используемым протоколом. Так, протокол http обычно использует порт 80.

- `more.html` — запрос к серверу. Обычно это имя нужного вам документа. К нему может быть указан путь, например `/archives/foo.html`, причем в конце запроса могут быть символы ? или `&`. Запрос сообщает серверу, что вы хотите от него получить.

Получив от пользователя URL, клиент выполняет следующие действия (рис. 17.2).

1. По имени сервера (`www.google.com`) находит соответствующий IP-адрес (`209.185.108.147`).
2. По IP-адресу и номеру порта устанавливается соединение с сервером.
3. У сервера запрашивается страница more.html. Клиент ожидает ответ.
4. Сервер посылает ответ, в данном случае — содержимое more.html, и закрывает соединение.
5. Клиент отображает содержимое ответа на экране.

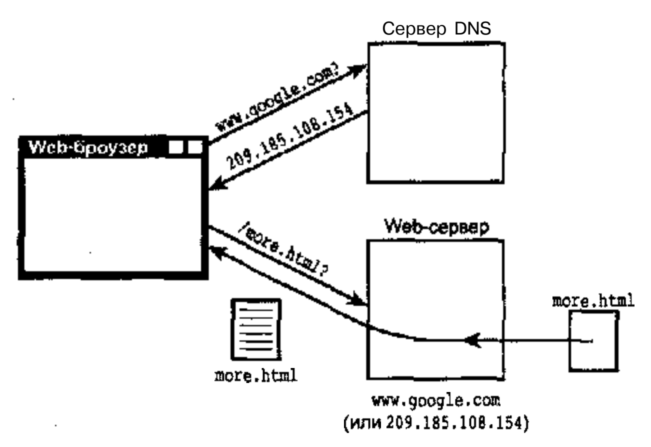


Рис. 17.2. Процесс запроса страницы

Подробное описание взаимодействия между клиентом и сервером приведено на 20-м занятии, "Работа с HTML-кодом и CGI-программами".

CGI и динамические Web-страницы

Во время загрузки обычной Web-страниды сервер находит на своем диске нужный документ и пересылает его клиенту (см. рис. 17.1).

Сервер, изображенный на рис. 17.1, никак не обрабатывает данные, он лишь анализирует запрос и передает требуемые данные клиенту.

Один из методов создания динамических Web-страниц предусматривает использование CGi-программ. CGI — общепринятый метод запуска на Web-сервере программ, генерирующих содержимое HTML-страниц. URL сообщает серверу, какая именно CGI-программа должна быть запущена, сервер ее запускает, она генерирует содержимое Web-страницы, и сервер пересылает это содержимое обратно клиенту, как показано на рис. 17.3.

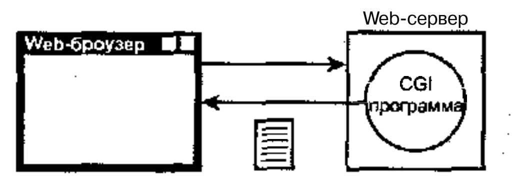


Рис. 17.3. Web-страница, сгенерированная CGI-программой

Во время каждого клиентского запроса страницы, являющейся продуктом выполнения CGI-программы, выполняются следующие действия.

1. Сервер запускает новый экземпляр CGI-программы.
2. Эта программа, используя необходимую информацию, генерирует страницу или другой ответ.
3. Содержимое страницы пересылается обратно клиенту.
4. CGI-программа завершает выполнение.

CGI-программа может быть написана на различных языках программирования, а не только на Perl (хотя мы только что и приступили к изучению использования для CGI-программирования сценариев на Perl). Тем не менее для этой цели может быть использован практически любой язык программирования: С, оболочка UNIX, Pascal, LISP, TCL. Тот факт, что многие CGI-программы написаны на Perl, — лишь счастливое совпадение. Perl прекрасно приспособлен для написания программ, работающих с текстом, а большинство CGI-программ предназначены для обработки и вывода текста.

Впрочем, CGI-программы могут выводить все, что угодно: изображения, текст в формате HTML, zip-файлы, видеопотоки и любой другой тип содержимого Web. Большинство CGI-программ предназначено для вывода текста в формате HTML.



CGI — это **не** язык, а протокол. Он не привязан ни к Perl, ни к HTML и лишь в малой степени зависит от протокола HTTP. CGI — согласованный интерфейс между Web-сервером и запускаемой на нем программой. Спецификация CGI содержится на Web-сервере Национального центра суперкомпьютерных приложений http://www.ncsa.uiuc.edu/cgi/interface.html. В последующих семи занятиях вы получите всю основную информацию, содержащуюся в этой спецификации.

Не пропустите этот раздел

Вы почти готовы к написанию CGI-программы. Но предварительно ознакомьтесь с дополнительной информацией по данному вопросу, иначе первый опыт CGI-программирования, скорее всего, окажется для вас неудачным. Это позволит вам сэкономить много времени и сил во время отладки.

Итак, для запуска и отладки CGI-программ нужен Web-сервер. Бич всех начинающих CGI-программистов — неправильно установленный Web-сервер. Получить доступ к Web-серверу можно несколькими способами: арендовать пространство на одном из коммерческих Web-серверов или установить свой собственный. Выбор из двух этих вариантов зависит от следующих факторов: сколько вы согласны платить, каковы требования к пропускной способности канала и насколько вы технически подкованы.

Если вы решили использовать коммерческий Web-сервер, найдите подходящий в Internet. Коммерческие серверы предоставляют услуги Web-хостинга. Их условия и расценки могут варьироваться в широких пределах. Перед написанием CGI-программы на Perl следует убедиться, что данный Web-сервер поддерживает Perl версии 5 в качестве языка для CGI-программирования. Лишь немногие компании, предоставляющие услуги Web-хостинга, не поддерживают Perl в качестве языка программирования CGI или вообще не поддерживают CGI. Не связывайтесь с ними — выбор достаточно велик и без них.

Удостоверьтесь также, что вам будет позволено писать собственные сценарии. Некоторые компании разрешают использовать лишь CGI-программы собственного производства, возможно даже, что за дополнительную плату. Подобных компаний также нужно избегать.

Существуют компании, которые берут дополнительную плату за обязательное тестирование ваших CGI-программ. Если вы остановили свой выбор на подобной компании, установите собственный Web-сервер для тестирования, потому что стороннее тестирование может влететь вам в копеечку.

Установить персональный Web-сервер несложно, для этого нужны минимальные технические знания и желание прочесть инструкцию по установке. Прежде всего нужно выбрать тип Web-сервера. Для операционной системы Windows существует по меньшей мере несколько десятков бесплатных или почти бесплатных программ Web-серверов. Нужно лишь выбрать тот из них, в котором Perl может использоваться для CGI-программирования. Для Windows также существует несколько коммерческих программ Web-серверов, наиболее известный из них — Microsoft Internet Information Server (IIS).

Для UNIX также написано несколько коммерческих Web-серверов. Их список можно получить у любого распространителя UNIX.

Наиболее популярный в Internet Web-сервер Apache — бесплатный. Web-сервер Apache легко установить при наличии компилятора С и им легко управлять, редактируя его файлы конфигурации. Apache можно установить и на Windows-системе. Дополнительную информацию по Apache можно найти по адресу http://www.apache.org.

Перед тем как запускать CGI-программу на персональном Web-сервере, убедитесь, что тот нормально обслуживает статические Web-страницы. Если Web-сервер не может обслужить обычные Web-страницы, сомнительно, что CGI-программы будут на нем работать.

Web-сервер нужно настроить таким образом, чтобы на нем можно было запускать CGI-сценарии. Отключенные средства запуска CGI — одна из головных болей начинающих CGI-программистов.

Контрольный список

Независимо от того, персональный у вас Web-сервер или вы арендуете пространство на коммерческом Web-узле, пройдитесь по всем пунктам приведенного ниже контрольного списка. Запишите все сведения, требуемые в нем, — позже это окажет вам неоценимую услугу.

- Если вы арендовали пространство на коммерческом Web-сервере, вам нужен доступ к необходимой информации. Она может находиться либо в разделе часто задаваемых вопросов Web-узла, либо в документации, посланной вам по электронной почте вместе с учетной записью. Если вы не можете найти эту информацию, обязательно свяжитесь с администратором Web-узла. Без нее правильная работа CGI невозможна.

- При самостоятельной установке и настройке Web-сервера нужная вам информация является частью процесса настройки. Для решения возникших проблем можно просмотреть соответствующие списки часто задаваемых вопросов и файлы конфигурации.

Вот эта *жизненно необходимая* информация для CGI-программирования.

- *Путь к интерпретатору Perl на Web-сервере.* Его нужно указывать после символов #1 в первой строке CGI-программы. Для Web-сервера, работающего в Windows, данная информация несущественна.
- *Расположение файлов системных журналов Web-сервера;* Вы не сможете легко отладить CGI-программы, не имея доступа к сообщениям об ошибках Web-сервера. Обязательно узнайте, где находятся эти файлы.
- *Расширение файлов для CGI-программ.* Некоторые Web-серверы определяют, что перед ними — статическая Web-страница или CGI-программа, по расширению файла. В CGI-файлах обычно используются расширения `.cgi` или `.pl`, а иногда он вообще не указываются.
- *Положение каталога CGI-программ.* Web-серверы могут идентифицировать CGI-программу или по расширению, или по ее положению в определенном каталоге. Сравнительно редко используются сразу два этих способа определения CGI-программы. Обычно CGI-каталог — это /cgi-bin, расположенный в корневом каталоге Web-сервера.
- *URL каталога CGI-программ.* Для запуска CGI-программы, кроме ее имени, нужно указать URL Web-сервера и имя каталога, CGI например http://www.myserver.com/cgi-bin/ или `http://www.myserver.com/cgi/`.

Первая CGI-программа

Только теперь, после всех наших предупреждений, контрольных списков и полезной информации, можете набрать вашу первую CGI-программу. Она приведена в листинге 17.1.

Наберите и сохраните эту программу в файле hello. Если согласно контрольному списку требуется определенное расширение имени файла — укажите его, т.е. если необходимо использовать расширение `.cgi`, назовите сценарий `hello.cgi`. Если же требуется расширение `.pl`, сохраните сценарий в файле hello.pl.

Вы же *действительно* выполнили все требования контрольного списка и получили необходимую информацию, не правда ли?

Листинг 17.1. Ваша первая CGI-программа

```
1:  #!/usr/bin/perl -w
2:  use CGI qw(:standard);
3:  use strict;
4:
5:  print header;
6:  print "<B>Hello, World!</B>";
```

 Проведем анализ программы.

- *Строка 1.* Это стандартная строка. Чтобы сценарий работал, *необходимо,* чтобы путь к интерпретатору Perl соответствовал тому, что вы записали при проверке контрольного списка. Ключ -w включает режим выдачи предупреждений.

- *Строка 2.* Модуль CGI, используемый программой. Оператор `qw(:standard)` определяет стандартный набор функций модуля, импортируемого в программу.
- *Строка 3.* Директива use strict ужесточает стилистический контроль. При этом не имеет значения, относится наша программа к CGI или нет.
- *Строка 5.* Из модуля CGI импортируется функция header. Она выводит стандартный заголовок, необходимый для обработки сервером информации, получаемой от CGI-программы.
- *Строка 6.* После вывода заголовка любая информация, выведенная CGI-программой, нормально отображается броузером. В нашем случае, если программа заработает, броузер выведет фразу Hello, World.

И это все? Ну, не совсем. Еще нужно поместить CGI-программу на сервер и протестировать ее. Пока же сделано лишь полдела.

Установка CGI-программы на сервер

Способ установки CGI-программы главным образом зависит от типа используемого сервера, наличия локального или FTP-доступа к нему и т.д. В следующих разделах описаны различные варианты установки CGI-программ.

Локальный доступ к Web-серверу в UNIX

Если вы можете войти в UNIX-систему Web-сервера с помощью программ telnet, rlogin или каких-нибудь других, воспользуйтесь следующими инструкциями.

1. Поместите CGI-программу `hello.cgi` (или hello.pl) на UNIX-сервер с помощью FTP. Можете сразу набрать программу в текстовом редакторе vi.
2. Переместите CGI-программу в нужный каталог с помощью команд `mv` или ср. Каталог вы должны были определить при проверке контрольного списка.
3. В UNIX необходимо сделать файл выполняемым. Для этого воспользуйтесь следующей командой:

   ```
   chmod 755 hello.cgi
   ```

   При необходимости вместо `hello.cgi` нужно подставить другое имя файла, например hello.pl. С помощью команды `chmod` устанавливаются права доступа к файлу. В нашем случае мы разрешаем изменять файл его владельцу, а читать и запускать — любому пользователю системы (так принято для CGI-программ).

ТОЛЬКО FTP-доступ к Web-серверу в UNIX

При наличии лишь FTP-доступа следуйте приведенным ниже инструкциям.

1. С помощью FTP-клиента поместите программу `hello.cgi` (или hello.pl) в каталог CGI-программ сервера. Вы должны были определить имя этого каталога при проверке контрольного списка. Пересылайте файлы в текстовом режиме или в режиме ASCII. Не пересылайте CGI-программу на сервер в двоичном режиме. В утилите FTP для пересылки файлов по умолчанию используется текстовый режим.

2. Сделайте CGI-программу выполняемой. В FTP для этого нужно набрать команду

   ```
   quote site chmod 755 hello.cgi
   ```

   Вместо hello.cgi используйте реальное имя файла программы. С помощью этой команды устанавливаются права доступа к файлу. В нашем случае мы разрешаем изменять файл его владельцу, а читать и запускать — любому пользователю системы.

3. Если у вас графический FTP-клиент, такой как Cute-FTP, вам нужно найти команду с названием Set Permissions, Change Mode, Set File Attributes или Set File Access Mode и изменить атрибуты файла с ее помощью.

4. Какой бы у вас ни был доступ, у владельца файла должны быть права на запись, чтение и выполнение файла, а у его группы и остальных пользователей — на чтение и выполнение. Если программа FTP-клиента требует числовое обозначение прав доступа, введите 755.

Локальный доступ к Web-серверу в Windows NT

Если у вас имеется локальный доступ к файловой системе Web-сервера в Windows NT, поместите CGI-программу в нужный каталог, имя которого вы должны были узнать при проверке контрольного списка. Для этого можно воспользоваться программой Проводник или любой другой утилитой копирования файлов.

ТОЛЬКО FTP-доступ к Web-серверу в Windows NT

При наличии лишь FTP-доступа к Web-серверу в Windows NT используйте FTP-клиент, чтобы поместить программу hello.cgi (или hello.pl) в каталог программ CGI сервера (его имя вы должны были узнать при проверке контрольного списка). Пересылайте файлы в текстовом режиме или в режиме ASCII. Не пересылайте CGI-программу на сервер в двоичном режиме. В утилите FTP для пересылки файлов по умолчанию используется текстовый режим.

Выполнение CGI-программы

Чтобы посмотреть, выполняется ли ваша программа, запустите броузер и укажите ему URL каталога CGI с именем программы, например:

```
http://www.myserver.com/cgi-bin/hello.pl
```

Вместо hello.pl введите реальное имя программы, например hello.cgi.

Произойдет одно из двух.

1. Броузер загрузит страницу с сообщением Hello, World.

2. Будет выведена страница с сообщением об ошибке.

Если CGI-программа по какой-то причине не работает — изучайте следующий раздел, который целиком посвящен проблемам отладки CGI-сценариев. Процедура установки и отладки CGI-программ *очень* сложная. Не прекращайте отладку программы, пока она не будет в полном порядке. После этого вам больше не придется с нею мучиться.

Если CGI-программа заработала, как и ожидалось, примите поздравления! Вы успешно установили Web-сервер, CGI-программу на нем и добились того, что она заработала. Все равно обязательно прочитайте следующий раздел, и вы узнаете, что делать, если CGI-программа не работает.

Что делать, если CGI-программа не работает

Перечисленные ниже разделы можно считать кратким руководством по отладке типичной CGI-программы. Перед тем как начать разбираться, почему не работает ваша первая CGI-программа, вернитесь к предыдущему материалу и проверьте, все ли инструкции выполнены. К концу этого занятия вы сможете локализовать любую проблему CGI-программы и исправить ее.

Все диагностические сообщения, приведенные в следующих разделах, предполагают, что отлаживаемая программа называется hello.cgi. Если у программы другое имя — используйте его.

А может, виновата сама программа?

Первый потенциальный источник проблем — конечно же, сама CGI-программа. Нечего мудрить с настройкой конфигурации Web-сервера, если в CGI-программе содержатся ошибки.

CGI-программы можно запускать вручную, как и остальные программы на Perl. Этот факт часто используется при отладке. Для запуска программы наберите в командной строке

```
perl hello.cgi
```

Интерпретатор Perl ответит вам подобной строкой:

```
(offline mode: enter name=value pairs on standard input)
```

Эта строка означает, что модуль CGI пытается получить значения полей формы. Про формы мы поговорим на 18-м занятии, "Основы обработки форм".

В ответ на это сообщение вы должны что-то ввести, хотя бы символ конца файла. В UNIX — это <Ctrl+D>, в Windows — <Ctrl+Z>. Затем CGI-сценарий выведет следующий текст:

```
Content-Type: text/html

<B>Hello, World!</B>
```

Сообщение Content-Type: text/html означает, что следующий за ним текст должен быть интерпретирован как текст HTML. Более основательно вы поймете значение этого сообщения после изучения материала 20-го занятия, "Работа с HTML-кодом и CGI-программами". Пока лишь вам достаточно знать, что эта *первая* строка необходима и что ее выводит функция header. Если перед этой строкой будет выведен какой-нибудь другой текст, CGI-программа не будет работать. Ниже перечислены несколько распространенных проблем, возникающих с CGI-программами, и способы их разрешения.

- **Проблема.** Сообщение о синтаксической ошибке.

  Решение. Устраните синтаксическую ошибку.

- **Проблема.** Сообщение Can't locate CGI.pm in @INC ___

  Решение. Вы используете неполную установку интерпретатора Perl. Модуль CGI устанавливается вместе с Perl. Для переустановки Perl воспользуйтесь инструкциями, приведенными в приложении "Инсталляция модулей в Perl".

Проблемы сервера

ЕСЛИ сценарий отлажен и все его ошибки исправлены, переходите к проверке правильности настройки сервера и установки сценария. Ниже перечислены несколько распространенных проблем, связанных с Web-сервером, и способы их разрешения.

- **Проблема.** Сообщение сервера Not found или 404 Not Found.

  Решение. Это сообщение свидетельствует о наличии одной из двух проблем.

  - Введен неправильный URL. Например, вы набрали https//www.server.com/cgi/hello.cgi вместо http://www.server.com/cgi-bin/hello.cgi. Снова вернитесь к контрольному списку и проверьте правильность указания URL и каталога CGI.
  - Сценарий помещен в неправильный каталог. Проверьте по контрольному списку правильность использованного каталога CG1 и при необходимости переместите сценарий в нужное место.

- **Проблема.** Отображается текст сценария.

  Решение. Это происходит потому, что Web-сервер воспринял файл программы как обычный документ. Причин может быть несколько.

  - CGI-программе назначено неправильное расширение, например вместо ,pl — .cgi или любое другое. Проверьте по контрольному списку, какое расширение нужно использовать в CGI-программах при работе с данным сервером.
  - Вы поместили сценарий в неправильный каталог и используете неправильный URL для доступа к нему. Убедитесь, что сценарий находится в требуемом каталоге и введен правильный URL.
  - Сервер неправильно сконфигурирован. Если вы используете собственный Web-сервер, внимательно прочтите документацию и проверьте корректность установок. Иногда в дистрибутив Web-сервера входит тестовый CGI-сценарий, попробуйте запустить его. Если вы работаете с коммерческим Web-узлом, убедитесь в том, что сценарий помешен в нужный каталог. При необходимости обратитесь за помощью в службу технической поддержки этого Web-узла.

- **Проблема.** Сообщение сервера Forbidden или 403 Error.

  Решение. Неправильно назначены права доступа к CGI-программе. Эта проблема характерна для Web-серверов, работающих под управлением UNIX.

  Для того чтобы посмотреть права доступа программы hello.cgi, перейдите в каталог, в котором она находится, и наберите в командной строке ls -1 hello.cgi. Эта команда позволяет отобразить права доступа в подобном виде:

  ```
  -rwxr-xr-x    1 user      93 Aug 03 23:06 hello.cgi
  ```

Права доступа — это символы -rwxr-xr-x. Если в вашем случае они не совпадают с приведенными, измените их в соответствии с приведенными выше инструкциями.

Устранение ошибок Internal Server Error или 500 Error

ЕСЛИ появляется сообщение Internal Server Error или 500 Error, это означает, что CGI-программа каким-то образом не смогла корректно выполнить свою работу. Это сообщение об ошибке общего назначения может быть получено во многих случаях.

Наиболее полезным средством для выявления ошибок, приводящих к этому сообщению, является файл системного журнала сервера. Все запросы Web-страниц заносятся сервером в этот специальный файл для дальнейшего анализа. При этом регистрируются все ошибки, включая ошибки при запуске CGI-программы.

Расположение файла системного журнала вы должны были определить при проверке контрольного списка. Обычно новые записи добавляются в конец файла системного журнала. Для просмотра в UNIX нескольких последних записей в файле журнала наберите в командной строке

```
tail server_log
```

Эта команда отобразит конец файла журнала. Некоторые Web-серверы имеют специальные утилиты для просмотра таких файлов. Часто такие утилиты представляют собой CGI-программы. Если у вас есть лишь FTP-доступ к Web-серверу, возможно, вам придется загрузить файл журнала на локальный компьютер и просмотреть его там.

Если у вас нет доступа к файлам журнала ошибок, дело совсем плохо. В этом случае вам придется устранять ошибку Internal Server Error методом последовательного исключения. Мы составили еще один контрольный список для поиска ошибок. Приведенные сообщения приблизительны и могут изменяться в зависимости от используемого сервера.

- **Запись в журнале.** No such file or directory: exec of /cgi-bin/hello.cgi failed

  Возможные причины.

  - Неправильная строка `#!` в сценарии. Убедитесь, что в этой строке указан корректный путь к интерпретатору Perl. Проверьте через FTP указанный каталог на наличие в нем интерпретатора Perl или воспользуйтесь командами `ls` или dir.
  - При пересылке CGI-программы на сервер через FTP использовался не ASCII-режим. Сценарии нельзя пересылать через FTP между Windows и UNIX в бинарном режиме.
  - Для CGI-программы в UNIX назначены неправильные права доступа. (См. описание ошибки Forbidden выше в разделе "Проблемы сервера".)

- **Запись в журнале.** Can't locate CGI.pm in @INC——

  Возможные причины.

  - У вас неполная установка интерпретатора Perl либо поврежденная, либо очень старая. Интерпретатор Perl не может определить положение модуля CGI, являющегося частью стандартной установки Perl. Переустановите Perl или попросите об этом системного администратора. Инструкции по установке модулей приведены в приложении к этой книге.

- **Запись в журнале.** Syntax error, warning, Global symbol requires, etc.

  Возможные причины.

  - В программе имеется опечатка или синтаксическая ошибка. Для их устранения следуйте инструкциям, приведенным в разделе "А может, виновата сама программа?".

- **Запись в журнале.** Premature end of script headers.

  Возможные причины. Это сообщение об ошибке охватывает широкий класс ситуаций, при которых сценарий первым выводит не заголовок Content-Type, а что-либо другое. Напомним, что этот заголовок выводится функцией header модуля CGI. Иногда в файле журнала ошибок может содержаться еще одна уточняющая запись перед или после указанной. Используйте ее для определения причины ошибки. Ваши действия в этом случае таковы.

  - Убедитесь, что до вызова функции header ваша программа не выводила никакой информации (включая сообщения об ошибках). Такой вывод перед функцией header приводит к возникновению ошибки.



> Некоторые CGI-программы не используют вызов функции header, а сами организуют первую строку вывода "Content-Type: text/html\n\n" . Многие считают, что такой способ и вызов функции header равносильны, но на самом деле это не так. На различных серверах последовательность символов \n\n может интерпретироваться по-разному, а функция header учитывает эти различия.

  - Проблема может быть связана с *буферизацией вывода*. При этом вывод функции system или оператора ` ` будет опережать вывод функции header. Для того чтобы избежать этого, перепишите начало программы таким образом:

```
#!/usr/bin/perl -wT
use strict;
use CGI;

$|=1;  # гарантирует вывод заголовка первым
print header;
```

Резюме

На этом занятии вы познакомились с основами работы CGI-программ. Вы узнали, чем отличаются между собой статические и динамические Web-страницы. На следующих занятиях будет приведена дополнительная информация по данному вопросу. Вы набрали вашу первую CGI-программу и, надеюсь, заставили ее работать.

Руководство по отладке CGI-программ, приведенное выше, пригодится вам на следующих занятиях.

Вопросы и ответы

У меня не установлен модуль CGI. Можно ли без него обойтись?

Пожалуй, нет. CG1 — достаточно сложный протокол. Многие опубликованные программы неудачно пытаются подменить функциональность модуля CGI. Они ненадежны с точки зрения проблем безопасности и совместимости, к тому же зачастую не следуют стандартам Internet. На 16-м занятии, "Сообщество Perl", вы узнали, почему не следует "изобретать велосипед". Тем более что CGI — это очень сложный "велосипед", и ни мне, ни вам невозможно изобрести его даже с сотой попытки.

Модуль CGI из стандартной установки Perl проверен сотнями и тысячами программистов и признан ими заслуживающим доверия. Используйте его.

В приложении описывается, как при необходимости установить модуль для личного пользования. Не существует причин, по которым нельзя использовать модуль **CGI.** Все примеры этой части требуют наличия установленного модуля CGI.

У меня есть копия cgi-lib.pl. Могу ли я использовать *этот* файл вместо модуля CGI?

Не нужно этого делать. Функциональность cgi-lib.pl полностью представлена в модуле CGI. Библиотека cgi-lib.pl на данный момент устарела и больше не поддерживается.

Почему обычно дня создания CGI-программ используют Perl, а, скажем, не C или TCL?

Некоторые особенности Perl делают его особенно полезным для создания CGI-программ. Вот краткий список этих особенностей.

1. В Perl имеются прекрасные средства для обработки текста.
2. Некоторые особенности Perl (о них вы узнаете позднее) позволяют создавать безопасные CGI-программы.
3. Perl — превосходный *интегрирующий* язык, позволяющий совместно использовать такие различные технологии, как утилиты операционной системы, средства доступа к базам данных и протокол CGI.
4. Perl удобен в использовании.

Могу ли я отправить вопрос, касающийся Perl и CGI, в группу новостей comp.lang.perl.misc?

Наиболее подходящая для этого группа новостей — comp.infosystems.www.authoring.cgi. Но вначале просмотрите список часто задаваемых вопросов, находящийся по адресу http://www.w3.org/CGI/.

Семинар

Контрольные вопросы

1. CGI-программа может быть написана с использованием:

   а) языка Perl, оболочки UNIX или языка C;

   б) только на языке Perl;

   в) на любом языке программирования, который может быть запущен на сервере.

2. Perl появился после Web?

   а) да;

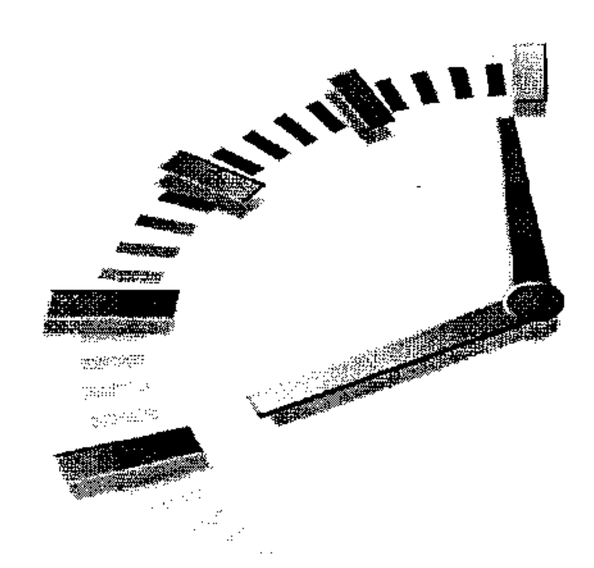
   б) нет.

Ответы

1. Правильный ответ — вариант в). Perl в этом смысле не уникален, просто на нем удобно создавать CGI-программы,
2. Правильный ответ — вариант б). Perl появился в 1987 году, а CERN разработал Web в 1991 году.

Упражнения

- Немного усложните CGI-программу "Hello, World!". Выведите текущее время с помощью функции localtime, добавьте цвет и таблицы с помощью дескрипторов HTML. Не бойтесь экспериментировать. Не забывайте, что выводимый программой текст HTML формирует Web-страницу, отображаемую броузером.



18-й час

Основы обработки форм

Наверняка во время работы в Internet вам приходилось заполнять различные формы HTML. Это могли быть формы регистрации, добавления в список рассылки, заказов, отзывов, опросов и многие другие. Все они используются для сбора всевозможной информации, например адресов электронной почты или настроек внешнего вида Web-узла.

Что же происходит, когда пользователь щелкает на кнопке Submit (Подать запрос)? Обычно после этого данные формы передаются в CGI-программу. На этом занятии вы узнаете, как передать данные формы в CGI-программу для их дальнейшей обработки.

Основные темы этого занятия.

- Основы обработки форм в CGI-программе на Perl.
- Отладка CGI-программ для обработки форм.
- Создание безопасных CGI-программ.

Как работают формы

Скорее всего, вы уже использовали формы, умеете их создавать и знаете, как они работают. Тем не менее мы освежим в вашей памяти информацию, касающуюся форм HTML.

Краткий обзор элементов форм HTML

Перед тем как разбираться, каким образом работают формы, вы должны вспомнить роль и назначение всех элементов форм.



Исходный код в формате HTML, представленный в этой книге, неполон. Для демонстрации возможностей CGI-программ мы использовали лишь часть стандарта HTML. Некоторые дескрипторы HTML, такие как <HEAD>, <BODY> или <DOCTYPE>, вообще не встречаются в наших примерах. Все копии экрана не содержат никаких графических элементов. Вы можете свободно добавить их по собственному усмотрению.

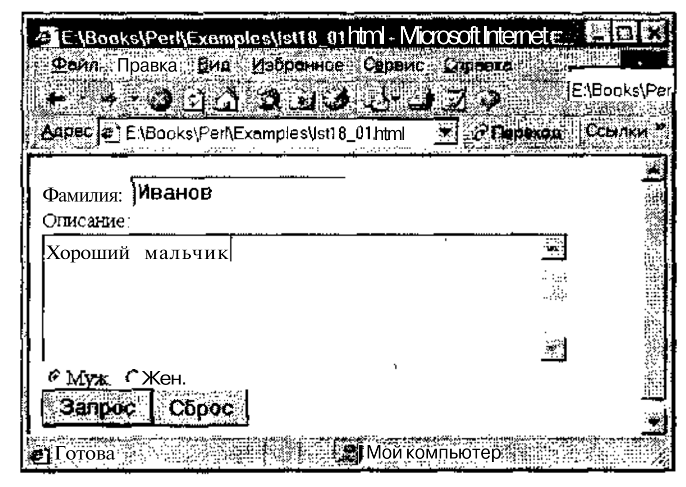
Форма HTML — часть документа HTML, служащая для организации ввода информации пользователем. При загрузке броузером Web-страницы с формой различные дескрипторы HTML создают на странице зоны пользовательского ввода. Пользователь может изменять состояние таких элементов, как флажки, переключатели, списки и текстовые поля. По окончании работы с элементами ввода в Web-броузере данные формы пересылаются CGI-программе для обработки.

В листинге 18.1 приведен код HTML типичной формы.

Листинг 18.1. Небольшая форма HTML

```
<FORM action="http://www.server.com/cgi-bin/submit.cgi" method="get">
Фамилия: <INPUT TYPE="text" name="name">
<BR>Описание:
<BR><TEXTAREA name="description" rows=5 cols=40>
</TEXTAREA>
<INPUT type="radio" name="sex" value="male">Муж.
<INPUT type="radio" name="sex" value="female">Жен.
<BR>
<INPUT type="submit" value="Запрос"><INPUT type="reset">
</F0RH>
```

На рис. 18.1 приведен вид этой формы в Web-броузере Internet Explorer 5.



Рис, 18.1. Вид формы, приведенной в листинге 18.1, в Web-броузере Internet Explorer 5.

Дескриптор <FORM> обозначает начало формы в документе HTML. Его атрибут method указывает, какой способ будет использован для передачи данных этой формы серверу — GET или POST. Если он отсутствует, для передачи данных формы CGI-программе используется метод GET. Разница между этими методами будет объяснена позже. Атрибут action определяет URL CGI-программы, получающей данные формы.

Дескриптор <INPUT> предоставляет пользователю область ввода, в данном случае пустое текстовое поле. Этому текстовому полю присвоено имя "name".

Дескриптор <TEXTAREA> отображается в броузере как поле для ввода нескольких строк текста. Его важнейший атрибут — name. Здесь его значение — "description". Каждый элемент формы HTML должен иметь свой атрибут name, не совпадающий с

подобными атрибутами остальных элементов формы. Когда данные формы поступают в CGI-программу для обработки, атрибуты name позволяют идентифицировать поля данных.

Элементы переключателя являются исключением из правила, согласно которому каждый элемент формы должен иметь уникальное имя. Переключатели могут объединяться в группы. В группе может быть выделен лишь один переключатель. У каждой группы переключателей должен быть собственный атрибут name.

Последний элемент формы — кнопка Submit. После щелчка пользователя на ней значения формы передаются в CGI-программу для дальнейшей обработки. Мы поговорим об этом в следующем разделе.

В спецификации HTML 4.0 предусмотрены и другие элементы формы, однако мы не будем приводить их полное описание на страницах данной книги. Многие элементы форм содержат различные атрибуты свойств, как, например, rows и cols в элементе TEXTAREA предыдущей формы. Во всех примерах мы будем использовать лишь основные атрибуты элементов.



С полной спецификацией HTML 4.0, содержащей асе допустимые элементы форм и их атрибуты, можно ознакомиться на Web-сервере http://www.w3c.org.

Что происходит после щелчка на кнопке Submit?

После того как пользователь заполнит в Web-броузере форму, происходит следующая последовательность событий.

- Данные формы группируются Web-броузером в пары имя—значение (рис. 18.2). Например, в приведенной форме поле name получает значение введенного пользователем текста. Поле sex получает значение выделенного переключателя. Эту работу выполняет Web-броузер.

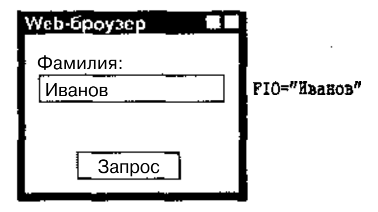


Рис. 18.2. Броузер группирует пары имен полей и их значений

- Устанавливается соединение с узлом, соответствующим указанному URL. Предположительно, это URL CGI-программы (рис. 18.3).

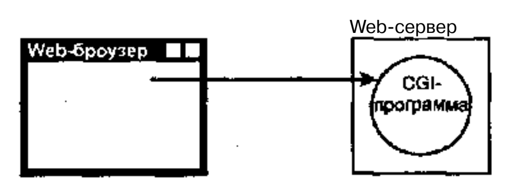


Рис. 18.3. Броузер устанавливает соединение с сервером

- Имена и значения полей формы передаются в CGI-программу с помощью одного из методов: GET или POST (рис. 18.4). Пока механизм этой передачи не должен вас волновать.

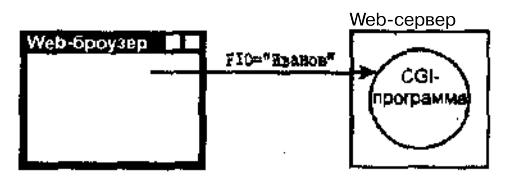


Рис. 18.4. Данные передаются серверу

- CGI-программа получает значения, генерирует ответ и посылает его обратно броузеру (рис. 18.5). Этим ответом может быть страница HTML, страница с другой формой, переадресация на другой URL или любой другой возможный вывод CGI-программы.

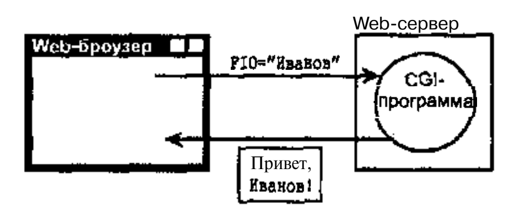


Рис. 18.5. CGI-программа Web-сервера возвращает ответ

Передача информации CGI-программе

CGI-программа, начавшая свое выполнение в результате передачи данных формы, должна обработать имена и значения полей, называемых *параметрами*. Для этого в модуле CGI имеется функция param.

Если вызвать эту функцию без аргументов, она возвратит имена полей, переданных CGI-программе. Если CGI-программе передается форма, HTML-код которой приведен в листинге 18.1, функция param возвратит список description, sex, name и submit.

Если у функции param имеется аргумент, она возвратит значение этого параметра. Например, вызов param('sex') возвратит значение переключателей — male или female, в зависимости от того, какой переключатель был выделен.

В листинге 18.2 приведена короткая CGI-программа для вывода этих параметров.

Листинг 18.2. CGI-программа для вывода параметров

```
#!/usr/bin/perl -w
use strict;
use CGI qw(:standard);

print header;
```

```
6: print "Фамилия: ", param('name'), "<BR>";
7: print "Пол: ", param('sex'), "<BR>";
8: print "Описание:<BR>", param('description'), "<P>";
```

Если параметр, указанный в аргументе функции param, не используется в форме, функция возвращает значение undef.

Методы GET и POST

В форме, текст которой приведен в листинге 18.1, у дескриптора <FORM> имеется атрибут method. Этот атрибут определяет, каким образом Web-броузер должен передавать данные Web-серверу. На данный момент существуют два метода.

Первый метод (он используется по умолчанию, т.е. в том случае, когда в дескрипторе <FORM> не указан атрибут method) — GET. При этом методе значения элементов формы передаются в CGI-программу в закодированном виде как часть URL. При работе в Internet часто можно видеть подобные URL:

```
http://www.server.com/cgi-bin/sample.pl?name=foo&desc=Basic%20Forms
```

CGI-программа преобразует последнюю часть URL и получит поля и их значения. Это происходит при вызове функции param. *Не* пытайтесь получить эти значения другим способом. Функция param идеально подходит для этого, и у вас едва ли возникнет необходимость воспользоваться другим кодом для извлечения значений полей.

Другой метод, POST, позволяет добиться того же результата, но другими средствами. При этом значения элементов формы не добавляются к URL, а пересылаются на стандартный вход CGI-программы после установки соединения с Web-сервером. Тонкости этого процесса вам знать не обязательно, в модуле CGI все они уже учтены. Как и раньше, для получения в программе значений полей достаточно вызвать функцию param.



В Internet или в других книгах можно встретить CGI-программы, которые проверяют переменную окружения QUERY_STRING или REQUEST_METHOD, чтобы определить, какой был использован метод — GET или POST. Эти программы пытаются воспроизвести функциональность стандартного модуля CGI и, вероятно, не столь успешно. Не делайте этого в ваших программах.

Итак, какой же метод выбрать? Каждый метод имеет свои преимущества и недостатки. Метод GET позволяет в Web-броузере создать закладку на URL, генерирующий данную страницу. Например, на URL

```
http://www.server.com/cgi-bin/sample.pl?name=foo&desc=Basic%20Forms
```

может быть сделана закладка и в дальнейшем в любой момент вызвана. Для CGI-программы sample.pl неважно, заполняли ли вы на самом деле форму или нет. Ей важно получить параметры для выполнения CGI-программы. Возможность многократного вызова CGI-программы при использовании метода GET называется *идемпотентностью.*

Возможно, вам не нужно, чтобы в Web-броузере можно было сделать закладку, непосредственно запускающую CGI-программу на вашем сервере. К тому же, следует заметить, что URL при использовании метода GET выглядит очень некрасиво.

Метод POST не использует URL для передачи данных, при этом методе данные пересылаются отдельно. Поэтому в данном случае закладку на страницу, генерируемую CGI-программой, сделать невозможно.

Основные сведения по вопросам безопасности в Web

Перед тем как поместить CGI-программу на Web-сервер, вы должны осознать, что тем самым вы предоставляете удаленным пользователям, использующим Web-броузеры, ограниченный доступ к вашей системе. В случае обычных документов HTML пользователи могут лишь загрузить статические документы. При использовании CGI-программ они смогут запускать программы на Web-сервере.

Написание безопасных CGI-программ сохранит нервы вам и администратору Web-сервера. Создавать такие программы несложно, нужно лишь следовать некоторым правилам.

Открытый канал

При загрузке в броузер страницы с Web-сервера коды HTML пересылаются по обычному каналу в открытом виде (рис. 18.6). Это означает, что эти данные не кодируются, не шифруются и не скрываются каким-либо другим способом.

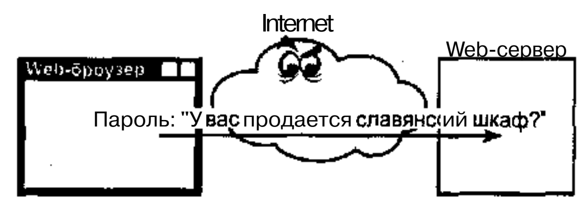


Рис. 18.6. Обычный текст пересылается на сервер

Данные, которые пользователь вводит при заполнении формы, передаются CGI-программе и пересылаются с использованием того же протокола, что и начальная Web-страница. Во время пересылки все поля формы открыты для всеобщего обозрения (рис. 18.7)



Рис. 18.7. Ответ сервера — обычный текст

Пересылка данных в открытом виде — одна из основных проблем, на которые вы должны обратить особое внимание. Internet — далеко не безопасное место. Кто угодно может подключиться к кабелю, по которому пересылаются данные между сервером и броузером, и перехватить их.

Учитывая все вышесказанное, вы никогда не должны передавать следующие данные посредством обычных CGI-форм.

- Любые пароли.
- Личную информацию (номера карточек социального страхования, телефонные номера).
- Финансовую информацию (номера счетов, PIN-коды, номера кредитных карточек).

Существует простое правило: никогда не пересылайте через Internet то, что нельзя смело доверить обыкновенной почтовой открытке.



> "Стоп!"— скажете вы. "Я же неоднократно встречался в Internet с формами, запрашивающими подобную информацию. Уверялось, что это совершенно безопасно". Достаточно безопасные транзакции могут быть осуществлены в Web с использованием дополнительных средств. Безопасные Web-транзакции предусматривают *полное* кодирование диалога между сервером и броузером. Для этого используется безопасная версия протокола http, называемая *https.*

Проверка данных на безопасность

Другой важный момент, на который нужно обратить внимание при написании безопасных CGI-программ, — зависимость выполнения команд Perl от информации, присылаемой от формы. В Internet и, возможно, в вашей внутренней сети встречаются люди, которых хлебом не корми, дай им только возможность повредить ваш Web-сервер. Даже обычные пользователи случайно могут послать неправильные данные CGI-программе и тем самым вывести ее из строя.

Посмотрите на код HTML формы и на CGI-программу, которые приведены соответственно в листингах 18.3 и 18.4.

Листинг 18.3. Форма для просмотра содержимого каталогов

```
<FORM action="/cgi-bin/directory.cgi">
Какой каталог просмотреть?
<INPUT TYPE=text NAME=dirname>
<INPUT TYPE=submit name=submit value="Запрос">
</FORM>
```

Листинг 18.4. Небезопасная CGI-программа `directory.cgi`

```
#!/usr/bin/perl -w
# НЕ используйте эту программу, она очень опасна
use strict;
use CGI qw(:all);

print header;
my $directory=param('dirname');
print `ls -l $directory`; # Просмотр каталога
```

В листинге 18.3 приведен короткий код формы, которая запрашивает имя каталога и передает его программе `directory.cgi`. Программа `directory.cgi` получает имя каталога и подставляет его в системную команду `ls -l`, эквивалент которой в DOS — команда dir. Эти команды выводят содержимое указанного каталога.

Программа такого рода позволяет удаленным пользователям исследовать структуру каталогов. CGI-программа не проверяет допустимость вводимого имени каталога, и броузер получает возможность доступа к любым конфиденциальным данным.

Наиболее интересно то, что в переменной $directory может вообще находиться не имя каталога! Например, если в поле dirname ввести значение /home; cat /ets/passwd, то выполняемая CGI-программой системная команда будет иметь вид

```
ls -l /home; cat /etc/passwd
```

Эта команда передаст копию файла системных паролей в Web-броузер. Фактически таким образом может быть запушена любая команда оболочки UNIX или MS-DOS. Если Web-сервер неправильно установлен и настроен, то это может сделать любой пользователь Internet.

В Perl имеется механизм, предотвращающий подобные вещи. Ключ -T, указанный в строке, начинающейся с #!, включает режим *контроля данных (data tainting)*. При этом данные, полученные из внешних источников (дескрипторов файлов, сетевых двунаправленных каналов, командной строки), помечаются как представляющие повышенную опасность. Помеченные таким образом данные нельзя использовать в системных вызовах, таких как open, функции system, системных командах и в других местах, чувствительных в смысле безопасности.

Функции open, system или оператор `` вообще нельзя использовать при включенном режиме контроля данных до тех пор, пока в программе явно не будет установлена переменная окружения PATH.

В листинге 18.5 приведен код более безопасной версии программы directory.cgi.

Листинг 18.5. Более безопасная версия программы **directory.cgi**

```
#!/usr/bin/perl -wT
# Включен режим контроля данных
use strict;
use CGI qw(:all);

print header;
# Явно указываются разрешенные пути
$ENV{PATH}='/bin:/usr/bin'
my $dir=param('dirname');
# Разрешено просматривать только каталоги,
#  находящиеся в каталоге /home/projects
if ($dir=~m,^(/home/projects/[\w/]+)$, ) {
  $dir=$1; # В этой переменной содержатся "чистые" данные,
           #  см. "perldoc perlsec"
  print `ls -1 $dir`;
}
```



Дополнительная информация по контролю данных и написанию безопасных программ на Perl содержится на странице persec электронного руководства, устанавливаемого вместе с Perl.

Невозможные события

Формы HTML/CGI могут вам доставить и другие неприятности. Посмотрите на фрагмент кода HTML, приведенный в листинге 18.6.

Листинг 18.6. Код HTML простой формы

```
<FORM action="/cgi-bin/doit.cgi">
Пожалуйста, назовете ваш любимый цвет:
<INPUT TYPE=text length=15 name=color>
<INPUT TYPE=submit value="Запрос">
</FORM>
```

В этой форме максимально возможная длина поля color составляет 15 символов, не правда ли? Теоретически, да. Спецификация HTML утверждает, что атрибут length обозначает максимально допустимую длину поля. Но броузер может быть взломан, или злоумышленник модифицирует форму так, чтобы можно было вводить большее количество символов.

Если вам нужно, чтобы некоторое поле имело определенное значение или длину, не полагайтесь при проверке только на код HTML, Java или JavaScript. Например, если длина поля color не должна превышать 15 символов, используйте в CGI-программе на Perl подобный код:

```
my $color=param('color');     # Получить исходное значение поля
$color=substr($color, 0, 15); # Взять только первые 15 символов
```

Отказ от обслуживания

Любой Web-сервер может быть перефужен слишком большим количеством запросов, поступивших от удаленных пользователей. Иногда это происходит по злому умыслу, а иногда — нет. Часто бывает так, что компания размещает свои службы в Web и затем из-за перегруженности сервера большим количеством запросов оказывается вынуждена их свернуть.

Ситуация отказа от обслуживания может произойти и с обычными Web-страницами, и с CGI-программами.

Вы мало что можете сделать для предотвращения проблемы отказа от обслуживания. Мощности Web-сервера должны быть адекватны количеству пользователей. Если CGI-программа слишком долго выполняется или использует слишком большое количество системных ресурсов из-за частого доступа к файлам, интенсивного использования центрального процессора, сервер становится уязвим для атак, приводящих к отказу от обслуживания. Поэтому постарайтесь максимально уменьшить и упростить CGI-программу.

Гостевая книга

В этом разделе в качестве примера мы рассмотрим настраиваемую гостевую книгу Web-сервера. Под гостевой книгой мы подразумеваем форму HTML, в которой пользователь оставляет свое имя, адрес и комментарии. Гостевая книга может использоваться для обратной связи по данной теме, для организации простой доски объявлений или для отправки вопроса в справочную службу. Все данные сохраняются в файле и могут быть выведены после заполнения формы; кроме того, они выводятся на собственной Web-странице.

В листинге 18.7 приведен короткий фрагмент кода HTML, представляющий форму гостевой книги абстрактной службы технической поддержки. Вы можете модифицировать эту форму таким образом, чтобы она соответствовала вашим потребностям.

Листинг 18.7. Форма для службы технической поддержки

```
<FORM action="/cgi-bin/helpdesk.cgi" name="helpdesk">
Тип проблемы:
  <INPUT TYPE=radio name=probtype value=hardware>
   Аппаратное обеспечение
   <INPUT TYPE=radio name=probtype value=software>
   Программное обеспечение
<BR>
<TEXTAREA name=problem rows=10 cols=40>
Опишите проблему.
</TEXTAREA>
<BR>
Ваше имя:
<INPUT TYPE=text width=40 name=name><BR>
<INPUT TYPE=submit name=submit value="Запрос">
</FORM>
```

Форму должна обрабатывать CGI-программа /cgi-bin/helpdesk.cgi. Текст этой программы представлен в листинге 18.8. Если вы хотите поместить программу в другое место или назвать ее как-нибудь иначе, измените URL в коде формы, приведенной в листинге 18.7. *

Листинг 18,8. CGI-программа службы технической поддержки

```
#!/usr/bin/perl -wT
use strict;
use CGI qw(:all);
use Fcntl qw(:flock);

# Путь к файлу журнала гостевой книги. Можете его изменить
my $gbdata="/tmp/guestbook";
# Для семафора может быть использовано любое имя.
my $semaphore_file="/tmp/helpdesk.sem";

# Функция блокировки (ожидает бесконечно долго)
sub get_lock {
  open(SEM, ">$semaphore_file")
      || die "Невозможно создать семафор: $!";
  flock(SEM, LOCK_EX) || die "Невозможно заблокировать файл: $!";
}
# Функция отмены блокировке файла
sub release_lock {
   close(SEM);
}

# Эта функция сохраняет данные формы в файле
sub save {
    get_lock();
    open(GB, ">>$gbdata")
             || die "Невозможно открыть файл $gbdata: $!";
   print GB "name: ", param('name'), "\n";
   print GB "type: ", param('probtype'), "\n";
```

```
   print GB "problem: ", param('problem'), "\n";
    close(GB);
    release_lock();
}
# Эта функция отображает содержимое файла гостевой книги
# в формате HTML с минимальным форматированием.
sub display {
   open(GB, $gbdata)
        || die "Невозможно открыть файл $gbdata: $1";
   while(<GB>){
        print "<B>$_</B><P>";  # ИМЯ
        my($type, $prob);
        $type=<GB>; # Каждая запись
        $prob=<GB>; # состоит из 3-х строк...
         print "$type<P>";
         print "$prob<BR><HR>";
   }
   close(GB);
}

print header;
# Параметр 'submit' передается этой CGI-программе,
# если она выполняется в результате нажатия кнопки
# 'submit' формы листинга 18,7
if (defined param('submit')) {
    save;
    display;
} else {
    display;
}
```



Большая часть кода листинга 18.8 должна быть вам понятна, обратите лишь внимание на следующее.

- Данная форма не может обойтись без функций `get_lock()` и `release_lock()`. Как вы знаете, одновременно может быть запушено несколько экземпляров одной CGI-программы. Это означает, что с файлом журнала может работать сразу несколько экземпляров программы `helpdesc.cgi`. Поэтому перед выполнением операции записи в файл его нужно заблокировать. Перед операцией чтения файл блокировать не нужно, поскольку чтение файла в тот момент, когда в него производится запись другим процессом, не вызывает никаких опасных последствий.

- Эта CGI-программа предусматривает два варианта использования. При вызове из формы, приведенной в листинге 18.7, она добавляет новые записи в файл журнала. При вызове не из формы — отображает содержимое файла журнала.

Резюме

На этом занятии вы изучили взаимодействие форм HTML и программ CGI. Мы показали, как CGI-программа может получать содержимое формы с помощью функции param модуля CGI. Мы также коснулись тем безопасного написания программ и регистрации данных. Вашему вниманию было предложено простое CGI-приложение гостевой книги. Оно может быть легко модифицировано и усовершенствовано для использования в ваших целях.

Вопросы и ответы

Моя форма не работает, я получаю сообщение об ошибке.

Для локализации проблемы воспользуйтесь руководством по отладке CGI, приведенном на 17-м занятии, "Введение в CGI". Отладка программы, обрабатывающей форму, не отличается от отладки обычной CGI-программы.

Я нашел в Internet прекрасную программу, но не могу понять, почему для получения параметров формы она использует переменную $ENV{QUERY_STRING}?

Потому что ее автор отказался от использования возможностей модуля CGI по обработке формы. Это означает, что или программа очень старая и была создана до появления этого модуля, или автор решил использовать собственный код обработки формы. В любом случае это свидетельствует о том, что к использованию данной программы нужно подходить с осторожностью.

Я запускаю программу с ключом -T в строке #!, при этом появляется сообщение об ошибке Too late for -т option и программа прекращает выполнение. Почему?

Ключ -T должен стать известен интерпретатору как можно раньше, чтобы Perl сразу мог начать процесс контроля данных. Иногда бывает, что к тому моменту, когда выполнение дойдет до строки программы #!, интерпретатор уже обработал параметры командной строки. Поэтому для запуска программы из командной строки, например во время отладки, необходимо в командной строке также указать ключ **-T:**

```
perl -T -d foo.cgi
```

Может ли контроль данных в Perl уберечь программиста от совершения нелепых ошибок в CGI-программе? Гарантированд ли теперь безопасность программы?

Ни одна CGI-программа не является абсолютно безопасной. Конечно, контроль данных поможет избежать многих нелепых ошибок, но не гарантирует полной безопасности программы.

Семинар

Контрольные вопросы

1. Что возвращает функция param без аргументов в контексте списка?

   а) undef;

   б) количество элементов формы;

   в) список имен элементов формы.

2. Есть ли различие для программиста между методами POST и GET при использовании модуля CGI?

   а) нет;

   б) да.

3. Обеспечивает ли безопасность поле ввода пароля в HTML-форме (пароль пересылается в скрытом виде)?

   а) нет;

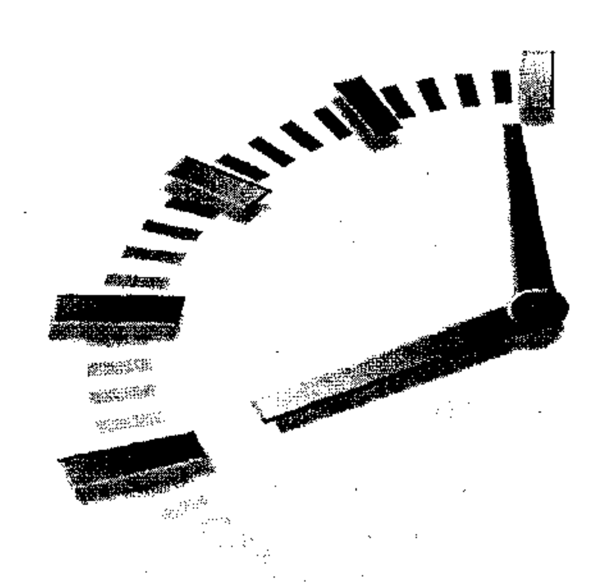
   б) да.

Ответы

1. Правильный ответ — вариант в). Функция param без аргументов возвращает список имен элементов формы, переданной CGI-программе.
2. Правильным будет вариант а). Никаких отличий нет.
3. Правильный ответ — вариант а). Нет. При использовании обычных протоколов **HTTP** и **CGI** все поля формы пересылаются в открытом текстовом виде. Поле ввода пароля лишь скрывает вводимый текст при наборе.

Упражнения

- Усовершенствуйте CGI-программу службы технической поддержки. Добавьте в файл журнала информацию о времени записи каждой записи и красиво оформите вывод.
- Функция display() выводит все записи в порядке их записи в файл журнала, начиная с самых старых. Измените эту функцию таким образом, чтобы вначале выводились самые свежие записи.



19-й час

Сложные формы

В Web встречаются не только простые одностраничные формы. Иногда форма может занимать сразу несколько страниц. Подобные варианты сложных форм часто используются в Web-приложениях, предназначенных для проведения опросов пользователей, электронной коммерции, справочных систем и т.д.

Для создания таких более сложных форм потребуется применение самых разных методов программирования, о которых и пойдет речь на этом занятии.

Итак, основная тема этого занятия — создание многостраничных форм.

Web-сервер "лишен памяти"

Обработка сложных многостраничных форм с помощью CGI-программ представляет собой уникальную задачу программирования. Связь между Web-броузером и Web-сервером достаточно кратковременна. Web-броузер подключается к серверу, считывает страницу, а затем отключается от него. Ни о какой поддержке долговременного соединения между сервером и броузером говорить не приходится.

Таким образом, при каждом следующем подключении Web-сервер не в состоянии распознать броузер в качестве абонента предыдущего сеанса связи. У сервера нет простого механизма запоминания подключающихся к нему броузеров.

Аналогией этой ситуации может служить разговор между посетителем библиотеки и библиотекарем, который лишен памяти, и посетителю во время визита разрешается задать только один вопрос.

Посетитель может заказать у библиотекаря книгу (скажем, об Аризоне), и библиотекарь, вполне вероятно, выполнит этот заказ, поскольку поиск одной книги — это достаточно простая задача. Но посетитель не может заказать другую книгу по той же тематике, поскольку библиотекарь ничего не помнит о предыдущем заказе. Если новый запрос прозвучит как "Дайте, пожалуйста, другую книгу об Аризоне", то библиотекарь, скорее всего, принесет ту же книгу, которую он выдал при первом заказе.

Единственный способ заказать вторую книгу по той же теме — сформулировать запрос следующим образом: "Мне нужна еще одна книга об Аризоне; у меня уже есть *Заселение юго-запада*". В этом запросе содержится достаточно информации для изложения сути проблемы, а также имеются сведения о том, какой вариант выполнения запроса нужно исключить.

При написании многостраничных форм для Web используется тот же подход: каждый сеанс вопроса/ответа должен содержать достаточно информации, чтобы Web-сервер мог "понять", что нужно сделать и что уже сделано. Такие сеансы можно создавать различными способами, и один из них — использование скрытых HTML-полей — как раз и представлен на этом занятии.

Скрытые поля

Простейший способ заставить Web-формы запоминать данные — поместить в них информацию с предыдущей формы с помощью скрытых полей. Скрытые поля являются частью спецификации HTML-форм. Они позволяют создавать поля, содержащие значения, которые являются частью HTML-формы, но при этом не отображаются в самой форме. На языке HTML они записываются следующим образом:

```
<INPUT type="hidden" name="fullname" value="Pink Floyd">
```

Если приведенный выше HTML-оператор поместить в форму, то новое имя (`"fullname"`) и значение ("Pink Floyd") станут частью этой формы. При передаче данных формы CGI-программе, написанной на языке Perl, название скрытого поля и его значение можно будет определить с помощью функции `param`.

Электронный магазин

В качестве примера использования скрытых полей рассмотрим электронный магазин, состоящий из набора Web-страниц и позволяющий делать покупки путем выбора элементов из электронного каталога. Пока же я просто покажу, как работает сложная форма, а ниже представлю другую сложную форму вместе с текстом программы создания формы для сбора информации.



> Не следует использовать пример рассматриваемого нами электронного магазина без реализации механизма безопасной Web-транзакции, о котором пойдет речь на 20-м занятии, "Работа с HTML-кодом и CGI-программами". Обратите внимание, что в этом примере мы не обрабатываем никакие персональные данные, типа номера телефона или кредитной карточки, поскольку скрытые поля подобны обычным полям HTML-формы. Здесь вообще нет и речи о какой бы то ни было безопасности.

Первая страница электронного магазина (рис. 19.1) содержит список особо рекламируемых товаров.

После того как пользователь щелкнет на кнопке Вход в магазин, CGI-программе будут переданы данные формы. В ответ на это CGI-программа выведет полный каталог, показанный на рис. 19.2.

Вторая страница представляет собой полный каталог товаров электронного магазина. После передачи данных первой страницы (с рекламируемыми товарами) CGI-программе последняя выведет новую форму с полным каталогом товаров. Кроме того, выбранные ранее товары будут помещены в эту же форму в виде скрытых полей.

При каждом получении CGI-программой значений из HTML-формы новая страница будет содержать старые значения в скрытых полях, а новые — в виде обычных элементов.

При такой организации взаимодействия с сервером можно избежать проблемы "забывчивого библиотекаря". При передаче серверу полного каталога, помимо сообщения о выборе элементов в текущей форме, скрытые поля "напоминают" CGI-программе о том, какие элементы были выбраны из первой формы.

Если необходима и третья страница, то в ней могут присутствовать значения из первых двух страниц, опять-таки в виде скрытых полей, как показано на рис. 19.3.

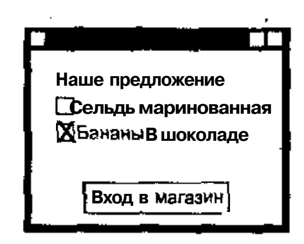


Рис. 19.1. Первая страница электронного магазина

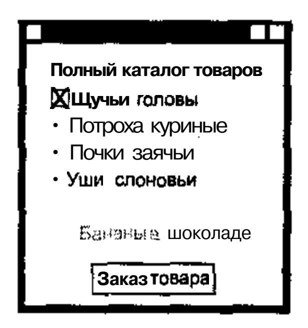


Рис. 19.2. Каталог электронного магазина

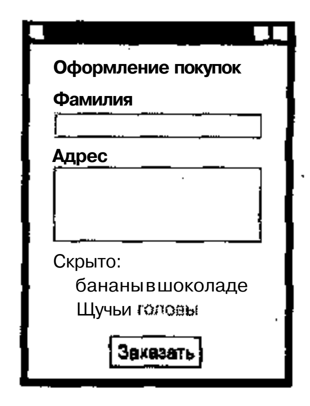


Рис. 19.3. Форма для оформления покупок

Следует знать о некоторых проблемах, сопровождающих использование скрытых полей на HTML-странице. Во-первых, значения в скрытых полях может увидеть кто угодно. Для этого пользователям достаточно просмотреть исходный HTML-код данной страницы, а в большинстве Web-броузеров такая возможность предусмотрена.

Во-вторых, удаленные пользователи при большом желании могут изменить значения в скрытых полях. Это можно сделать, используя специальный Web-броузер или передавая данные формы вручную с помощью протокола HTTP. Поэтому при создании электронного магазина не следует хранить цены на товары в скрытых полях. Для идентификации товаров и цен используйте специальные коды. Учтите, что CGI-программа должна "заглядывать" в ценник только при отображении списка товаров.



При разработке форм имеет смысл поинтересоваться, как это делают другие, и перенять некоторый опыт использования скрытых полей для сохранения информации. При этом следует воспользоваться возможностью просмотра исходного кода страницы, которая предоставляется большинством Web-броузеров в виде команды View⇨Page Source. (В броузере Internet Explorer эта команда называется В виде HTML.) И не забывайте, что перенимать опыт можно лишь на уровне идей: вы не должны копировать чужую форму, поскольку копирование означает нарушение авторских прав.

Многостраничная форма для сбора информации

Проведение опросов и сбор информации в Internet — прекрасные примеры многостраничных форм. Как правило, объем содержимого таких форм слишком велик, чтобы вся информация смогла уместиться на одной странице, поэтому его делят на несколько страниц по некоторому принципу, например по категориям.

Ниже рассматриваются примеры форм для проведения простого опроса, предназначенного для выяснения некоторых аспектов индивидуальности пользователей. Этот опрос состоит из четырех различных форм, но при желании их число можно легко изменить. Итак, вот как распределены функции между этими четырьмя формами.

1. Ряд вопросов общего характера, которые иногда используются для выяснения типа личности.

2. Некоторые конкретные вопросы о привычках и вопрос, основанный на ответах, содержащихся на первой странице исследования.

3. Форма, позволяющая ввести имя и комментарии относительно данного исследования.

4. Сообщение, содержащее благодарность, которое выводится по завершении исследования.

Для реализации всех четырех функций используется одна и та же CGI-программа. Именно она (в зависимости от номера текущей страницы) принимает решение о том, какая страница будет открыта следующей. Ядро программы представлено в листинге 19.1.



> Включение в текст программы директивы
>
> use `CGI::Carp qw(fatalsToBrowser);`
>
> заставляет CGI-программу выдавать сообщения, являющиеся частью Web-страницы, но которые обычно относятся к файлу системного журнала Web-сервера. При написании больших CGI-программ подобные директивы могут облегчить процесс отладки.

Результаты опроса сохраняются в текстовом файле, но не отображаются этой программой. Данная программа просто собирает ответы и сохраняет их в файле для дальнейшего использования. А для отображения результатов вам придется написать другую CGI-программу.

Листинг 19.1. Первая часть CGI-программы, выполняющая сбор информации

```
1:  #!/usr/bin/perl -w
2:  use Fcntl qw(:flock);
3:  use CGI qw(:all);
4:  use CGI::Carp qw(fatalsToBrowser);
5:  use strict;
6:  my $surveyfile="/tmp/survey.txt";
7:  my @survey_answers=qw(pettype daytype clothes
8:             castaway travel risky ownpet
9:             realname comments);
10: my $semaphore_file="/tmp/survey.sem";
11: print header;
12: if (! param ) {
13:        page_one();    # Опрос только начался
14:   } elsif (defined param('pageone')) {
15:        page_two();    # Получены ответы на первую страницу,
                          # вывод второй
16:   } elsif (defined param('pagetwo')) {
17:        page_three();  # Вывод последней страницы.
18:   } else {
19:        survey_done(); # Вывод благодарности и сохранение данных
20:   }
```

Анализ Проведем анализ программы.

- *Строки 6–8.* Во время проведения опроса каждая HTML-форма содержит поля ввода. Имена всех полей собраны в используемом здесь массиве. Этот массив имен позже будет использован в функциях `save()` и `repeat_hidden()`.

- *Строки 12–13.* Если эта CGI-программа вызывается без параметров, т.е. не для обработки данных формы, вызывается функция page_one(), предназначенная для вывода первой формы для проведения опроса.
- *Строки 14–17.* Если этой CGI-программе передается параметр HTML-формы, именуемый **pageone**, вызывается функция **page_two()**. Если же передается параметр **pagetwo**, вызывается функция page_three().
- *Строка 19.* Если этой CGI-программе передаются параметры HTML-формы, но они не являются параметрами pageone или pagetwo, то это означает, что нужно завершить выполнение опроса. В этом случае с помощью функции **survey_done()** сохраняются результаты опроса, и его участнику выводится сообщение с благодарностью.

На каждой странице предусмотрены кнопки, обеспечивающие загрузку следующей формы, как показано на рис. 19.4. Поскольку имя кнопки передается CGI-программе как параметр, его можно использовать для отображения номера страницы, данные которой были только что переданы программе.

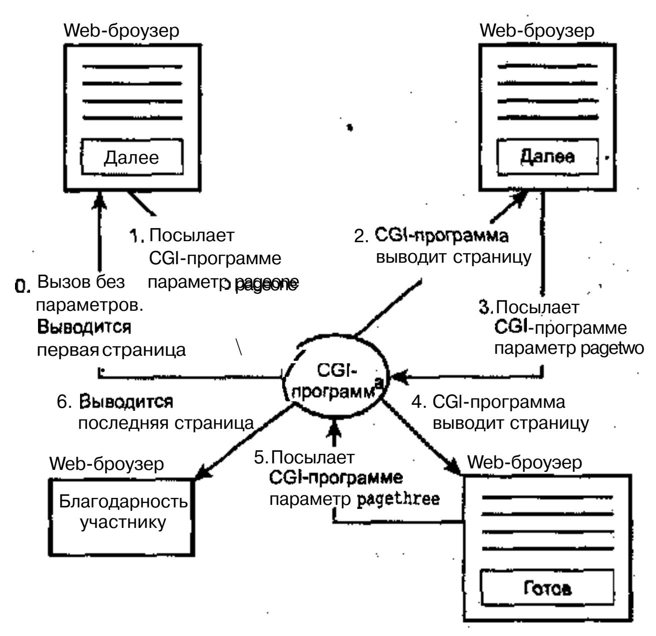


Рис. 19,4. Схема действий, выполняемых в результате использования кнопок на страницах HTML-формы

В листинге 19.2 вы найдете продолжение CGI-программы для проведения опроса.

Листинг 19.2. Вторая часть CGI-программы, выполняющей сбор информации

```
21: sub page_one {
22:     print<<END_PAGE_ONE;
23: <FORM>
24: Вы любите кошек или собак?<BR>
25: <INPUT type=radio name=pettype value=dog>Собак<BR>
26: <INPUT type=radio name=pettype value=cat>Кошек<BR>
```

```
27:   <P>
28:   Вы относите себя к жаворонкам или к совам?<BR>
29:   <INPUT type=radio name=daytype value=early>К жаворонкам<BR>
30:   <INPUT type=radio name=daytype value=late>К совам<BR>
31:    <P>
32:    Если бы вы могли сами решать, как одеваться на работу——<BR>
33:    <INPUT type=radio name=clothes value=casual>В свободном стиле<BR>
34:     <INPUT type=radio name=clothes value=business>В деловом стиле<BR>
35:     <P>
36:    Если бы вы вдруг очутились на безлюдном острове,
37:    какую компанию вы бы предпочли?<BR>
38:    <INPUT type=radio name=castaway 30:   value=ginger>Василий Теркин<BR>
39:    <INPUT type=radio name=castaway value=marya>Дева Мария<BR>
40:    <INPUT type=radio name=castaway value=prof>Профессор<BR>
41:     <INPUT type-radio name=castaway value=skipper>Капитан корабля<BR>
42:     <INPUT type=submit name=pageone value="Далее">
43:      </FORM>
44:    END_PAGE_ONE
45:     }
```

- *Строки* ***22—44*** представляют собой новую конструкцию языка Perl, с которой вы еще не встречались. Она называется *встроенным документом* (inline document). Эта конструкция позволяет определить текстовую строку, которая служит признаком конца документа в программе. Примечательно то, что встроенные документы можно помещать прямо в программы на Perl, располагать их на нескольких физических строках, включать в них переменные Perl, которые замещаются как и в обычных строках, заключенных в двойные кавычки. Чтобы начать встроенный документ, используйте символы «, за которыми должно следовать любое слово или текстовая строка. Документ продолжается до тех пор, пока не встретится следующее вхождение указанного слова, расположенного в начале строки, как показано в следующем примере:

  ```
  $a=<<END_OF_QUOTE;
  Все это считается встроенным документом.
  END_OF_QUOTE
  ```

  За словом, идентифицирующим начало встроенного документа (в предыдущем примере — это строка `END_OF_QUOTE`, а в листинге 19.2 — `END_PAGE_ONE`), должна стоять точка с запятой. В конце встроенного документа это слово должно начинаться с первой колонки и не должно иметь "в хвосте" никаких символов типа пробелов или точек с запятой. Внутри встроенного документа переменные ведут себя так же, как и в обычных строках, заключенных в двойные кавычки ("")- Таким образом, во встроенных документах нужно осторожно использовать символы $ и @.

  С помощью встроенных документов в программы на Perl можно встраивать большие фрагменты HTML-кода, что избавит вас от необходимости возиться с символами двойных кавычек, множеством операторов PRINT и т.п.

Функция, представленная в листинге 19.2, просто выводит HTML-форму. В дескрипторе `<FORM>` не указан какой бы то ни было метод или сценарий обработки данных формы. Поэтому при нажатии кнопки запроса сервер повторно запустит ту же CGI-программу, которая сгенерировала форму. Если не задан атрибут method, используется стандартный метод GET.

Обратите внимание, что кнопке запроса в форме присвоено имя pageone. При передаче данных этой формы CGI-программе будет послан параметр с именем pageone; в данном случае его значение не важно. Именно сам передаваемый параметр является определяюшим фактором для загрузки CGI-программой второй страницы.

Следующий фрагмент CGI-программы сбора информации приведен в листинге 19.3.

Листинг 19.3. Третья часть CGI-программы, выполняющей сбор информации

```
46:   i Помещает ответы пользователя в скрытые поля
47:    sub repeat_hidden {
48:        foreach my $answer ( @survey_answers ) {
49:            if (defined param($answer)) {
50:                 print "<INPUT TYPE=hidden";
51:                print " name=$answer ";
52:                print " value=\"", param($answer),"\">\n";
53:            }
54:        }
55:   }
56:   sub page_two {
57:       my $pet=param{'pettype');
58:      if (! defined $pet) {
59:           $pet="goldfish";
60:       }
61:       print<<END_PAGE_TWO;
62:   <FORM>
63:   Вы бы предпочли...<BR>
64:   <INPUT type=radio name=travel value=travel>Путешествовать<BR>
65:   <INPUT type=radio name=travel value=home>Оставаться floua<BR>
66:   <P>
67:   Вы считаете себя...<BR>
68:   <INPUT type=radio name=risky value=yes>Безрассудным<BR>
69:   <INPUT type=radio name=risky value=no>Осторожным<BR>
70:   <P>
71:    У вас есть $pet?<BR>
72:    <INPUT type=radio name=ownpet value=$pet>Да<BR>
73:    <INPUT type=radio name=ownpet value=no>НеТ<BR>
74:    <P>
75:    <INPUT TYPE=submit name=pagetwo value="Далее">
76:   END_PAGE_TWO
77:       repeat_hidden();
78:       print "</FORM>";
79:   }
```

- *Строка 47.* Как ясно из комментария в строке 46, функция в этой строке помешает значения всех полей предыдущей формы в скрытые поля текущей формы. Массив @survey_answers содержит все возможные значения атрибута "name=" в HTML-формах. При первом выполнении CGI-программы большинство из этих полей еще не определено, поскольку относящиеся к ним части опроса еще не проведены.

- *Строки 48–49.* Проверяются все возможные параметры формы, которые перечислены в массиве @survey_answers. Если параметр определен, то его значение помещается в скрытое поле. При этом используется дескриптор HTML <INPUT TYPE=hidden>.

- *Строки 56~60.* Эта функция вызывается для вывода второй страницы формы. Если первая страница была заполнена правильно, параметр param('pettype') будет иметь значение dog или cat, которое помешается в переменную `$pet`. Если же участник исследования проигнорирует этот вопрос и параметр `param('pettype')` останется неопределенным, будет использовано стандартное значение goldfish.
- *Строки 61~76.* Выводится остальная часть второй страницы формы, и параметр $pet подставляется в вопрос. Таким образом, данный вопрос зависит от ответов пользователя, которые он сделал на первой странице.
- *Строка 77.* Все параметры HTML-формы из первой страницы переносятся в эту форму как скрытые поля.

Если проанализировать форму на данном этапе (на момент формирования второй страницы), то окажется, что все ответы из первой страницы хранятся как скрытые поля в конце второй страницы формы. Программа вывода третьей страницы формы представлена в листинге 19.4.

Листинг 19.4. Четвертая часть CGI-программы, выполняющей сбор информации

```
80: sub page_three {
81:       print<<END_PAGE_THREE;
82:  <FORM>
83:  Последняя страница!  Эта информация необязательна!<BR>
84:  Ваше имя:
85:  <INPUT TYPE=text name="realname"><BR>
86:  Ваши предложения и замечания:<BR>
87:  <TEXTAREA NAME=comments cols=40 rows=10>
88:  </TEXTAREA>
89:  <P>
90:  <INPUT TYPE=submit name=pagethree
91:      value="Submit survey results">
92: END_PAGE_THREE
93:       repeat_hidden();
94:       print "</FORM>";
95: }
```

Функция page_three довольно проста: она лишь выводит в форме текстовое поле и сообщение. А в конце снова вызывает функцию `repeat_hidden()`, чтобы поместить все скрытые поля на третью страницу формы. Последняя часть CGI-программы представлена в листинге 19.5.

Листинг 19.5. Последняя часть CGI-программы, выполняющей сбор информации

```
96:  sub survey_done {
97:       save();
98:       print "Спасибо!";
99:  }
100: #
101: # Все результаты исследования сохраняем в файле $surveyfile
102: #
103: sub save {
104:      get_lock();
```

```
105:       open(SF, ">>$surveyfile")
106:          || die "Ошибка при открытии файла $surveyfile: $!";
107:       foreach my $answer (@survey_answers) {
108:           if (defined param ($answer) ) {
109:               print SF $answer, "=", param($answer), "\n";
110:           }
111:       }
112:       close(SF);
113:       release_lock();
114: }
115:     # Чтобы несколько участников не могли одновременно
116:     # записать информацию в файл базы данных,
117:     # мы будем его блокировать в момент записи.
118:     #
119:
120:   # Функция блокировки (ожидает неограниченно долго)
121:   sub get_lock {
122:      open(SEM, ">$semaphore_file")
123:         || die "Ошибка при создании семафора: $!";
124:      flock(SEM, LOCK_EX) || die "Блокировка не выполнена: $!";
125:   }
126:
127:   # Функция снятия блокировки
128:   sub release_lock {
129:      close(SEM);
130:   }
```

- *Строка 96.* Данная функция вызывается для вывода слов благодарности за участие в опросе. Это всегда стоит делать, особенно если анкета занимает несколько страниц. Для сохранения данных в файл вызывается функция save().
- *Строка 103.* Функция save() — это практически двойник функции save(), описанной на 18-м занятии, "Основы обработки форм". Она блокирует файл базы данных с помощью функции get_lock(), записывает в него ответы пользователя с помощью фрагмента, аналогичного используемому в функции repeat_hidden(), а затем снимает блокировку с файла путем вызова функции release_lock{).

Описанную выше программу сбора информации вы можете изменять по своему усмотрению. Ее дизайн достаточно гибок, что позволяет выполнить адаптацию программы для самых разных целей.

Резюме

На этом занятии вы узнали, как выполняется обработка данных многостраничных Web-форм, а также о проблемах, которые приходится при этом решать. Самая серьезная из них — запоминание на каждой последующей странице данных, собранных на предыдущих страницах. Здесь вы узнали, что для запоминания на Web-страницах информации, которую не в состоянии запомнить сервер, используются скрытые поля.

Вопросы и ответы

Почему формы, рассмотренные на этом занятии, эстетически некрасивы?

Формы, приведенные в этой книге, довольно просты, иногда схематичны, но кому-то они могут показаться неэстетичными. Цель книги — научить читателя программированию на Perl и созданию CGI-программ, а не искусству HTML-дизайна. Действительно, большинство приведенных здесь HTML-фрагментов не отвечают общепринятым стандартам и не доведены до конца. Во многих из них отсутствуют дескрипторы <HEAD>, <HTML>, заголовки и т.п. Надеюсь, вы внесете соответствующие коррективы в соответствии с вашими потребностями. Как упоминалось выше, если хотите внести изюминку в свои формы — исследуйте просторы Web и найдите на них те формы, которые отвечают вашим вкусам. Просмотрев исходный код понравившейся формы, можно понять идею ее построения, а затем воспользоваться ею для создания собственных форм.

Что означает такое сообщение об ошибке: `Can't find string terminator "XXXX" anywhere before EOF at ....`

Эта ошибка вызвана присутствием в вашей программе открывающих кавычек и отсутствием закрывающих. При использовании встроенных документов это сообщение означает, что не найдено слово, использованное в качестве признака конца документа. Формат встроенных документов имеет следующий вид:

```
print «MARK;
текст
текст
текст
HARK
```

Начальное и конечное слово встроенного документа — в данном случае это слово MARK — должно быть одним и тем же. Причем завершающему слову не должно ничего ни предшествовать, ни следовать за ним — оно должно быть единственным на строке. Текстовые редакторы MS-DOS и Windows иногда не помещают символ конца строки после последней строки программы. Если у вас завершение встроенного документа совпадает с концом файла, попробуйте после этого документа поместить пустую строку.

Семинар

Контрольные вопросы

1. Чтобы ваша программа помнила длинные многостраничные Web-транзакции, необходимо использовать:

   а) базы данных и файлы cookie;

   б) скрытые поля HTML-формы;

   в) определенную комбинацию скрытых полей HTML-формы, файлов cookie и базы данных.

2. Использование дескриптора HTML <FORM> без атрибута action:

   а) не будет работать;

б) заставит сервер использовать для обработки данных формы CGI-программу, которая первоначально сгенерировала эту страницу;

в) заставит сервер перезагрузить текущую страницу.

3. Представленная на этом занятии CGI-программа сбора информации содержит небольшую ошибку. В чем она Заключается?

а) оператор print <<EOP; в листинге 19.2 имеет неверный синтаксис;

б) HTML-код не завершен, поскольку в нем отсутствует дескриптор <HEAD>;

в) программа сбора информации не выводит никаких результатов.

Ответы

1. Правильным вариантом будет либо б), либо в). Можно использовать только скрытые поля или только файлы cookie. Использование только базы данных не даст нужного результата.

2. Правильный ответ — вариант б). При перезагрузке текущей страницы будут очищены все поля текущей формы. Если в дескрипторе <FORM> отсутствует атрибут action, для обработки данных формы сервер использует URL текущей страницы, т.е. URL CGI-сценария, который вывел первую страницу формы.

3. Правильный ответ — вариант б). Оператор print «EOP; имеет вполне допустимый синтаксис и определяет начало встроенного документа. Вариант в) не подходит, поскольку в программе это и не предусмотрено (см. раздел "Упражнения").

Упражнения

- Напишите короткую CGI-программу для отображения результатов работы программы сбора данных. Возможно, следует построить таблицу, отображающую результаты следующим образом.

Кошка/собака	Имеете ли его дома?	Сова	Манера одеваться	После кораблекрушения	Путешественник	Готовность к риску
Кошка	Нет	Да	Свободная	Профессор	Да	Да
ни то ни другое	Рыбки	Нет	Деловая	Капитан	Нет	Нет
Собака	Да	нет	Свободная	Дева Мария	Да	Да

- Для усложнения задачи напишите CGI-программу, в которой подводятся итоги по результатам опроса и выводятся, например, в таком виде:

Любители кошек/собак:	**кошек 40%**	**собак 45%**	**ни то ни другое 15%**	
Имеют животное дома:	кошек 20%	собак 15%	рыбок 30%	никого 35%
Сова?	да 35%	нет 40%		

20-й час

Работа с HTML-кодом и CGI-программами

На этом занятии вы узнаете, как с помощью CGI-программ можно сделать свой Web-узел более гибким и управляемым.

Основные темы этого занятия.

- Каким образом HTML-код попадает из сервера к вашему броузеру.
- Как CGI-программы посылают что-либо, отличное от HTML-страниц.
- Как напрямую передать значения CGI-программе.
- Как работают серверные включения (SSI).
- Как получить информацию о броузере и сервере.

Протокол HTTP

На 17-м занятии, "Введение в CGI", мы уже говорили о том, как осуществляется взаимодействие между Web-броузером (таким как Netscape или Internet Explorer) и Web-сервером (например, Apache или IIS) с помощью протокола CGI. Рассматриваемый нами процесс был несколько упрощен. Теперь, после того как вы узнали, что такое CGI, пришло время разобраться с протоколами взаимодействия броузера и сервера более подробно. Чуть позже на этом же занятии вы познакомитесь с некоторыми методами управления этим взаимодействием, позволяющими решать ряд интересных задач.

Упомянутое выше взаимодействие сервера и броузера описывается специальным протоколом, который называется *протокол передачи гипертекста (Hypertext Transfer Protocol — HTTP)*. В настоящее время применяются две версии этого стандарта: HTTP 1.0 и HTTP 1.1 (для обсуждаемых ниже вопросов подходит любая из них).



Документы стандартов, в которых описаны протоколы, используемые в internet, называются Request For Comments, или RFC. Эти документы, поддерживаемые организацией Internet Engineering Task Force (IETF), можно просмотреть в Web по адресу `http://www.ietf.org`. Протокол HTTP описан в документах RFC 1945 и RFC 2616. Однако имейте в виду, что эти документы рассчитаны на подготовленных пользователей.

Когда ваш Web-броузер устанавливает соединения с Web-сервером, броузер посылает серверу начальное сообщение, которое выглядит следующим образом:

```
GET http://testserver/ HTTP/1.0
Connection: Keep-Alive
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Charset: iso-8859-1, *,utf-8
Accept-Encoding: gzip
Accept-Language: en, en-GB, de, fr, ja, ko, zh
Host: testserver:80
User-Agent: /4.51 [en]C-c32f404p (WinNT; U)
```

По строке GET можно судить о том, с какого адреса URL вы пытаетесь получить документ и какую версию протокола используете. В данном случае вы используете версию 1.0 протокола HTTP.

Строка Connection означает, что вы хотели бы оставить это соединение открытым для получения нескольких страниц сразу. По умолчанию броузер создает отдельное соединение для каждого фрейма, страницы и изображения на Web-странице. Директива Keep-Alive просит сервер поддерживать соединение открытым, чтобы можно было принимать несколько элементов, используя одно и то же соединение.

Строки Accept определяют, какие виды данных вы хотели бы принимать с помощью этого соединения. Символы */* в конце первой строки Accept означают, что вы не прочь принимать любые виды данных. Следующая строка (iso-8859-1 и остальные) определяет, какое кодирование символов может быть использовано для документа. Строка Accept-Encoding: gzip в данном случае означает, что для сжатия данных, получаемых от сервера, с целью их быстрой передачи может быть использована утилита gzip (GNU Zip). Наконец, строка Accept-Language говорит о том, какие языки приемлемы для этого броузера: английский (США), английский (Великобритания), немецкий, французский и т.д.

В строке Host указывается имя сервера, обслуживающего Web-узел. Благодаря виртуальности обслуживания (пояснения ниже) это имя может отличаться от имени компьютера в URL.

Наконец, броузер идентифицирует себя для Web-сервера как Mozilla/4.51 [en]C-c32f404p (WinNT; U). В Web-терминологии броузер называется *пользовательским агентом* (user agent).

Затем сервер посылает броузеру ответ, который выглядит примерно так:

```
GET http://testserver/ ~> 200 OK
Date: Thu, 02 Sep 1999 19:54:39 GMT
Server: Netscape-Enterprise/3.5.1G
Content-Length: 2222
Content-Type: text/html
Last-Modified: Wed, 01 Sep 1999 17:12:03 GMT
```

За ответом следует содержимое запрашиваемой вами страницы.

Строка GET в данном случае означает, что сервер собирается передать броузеру Web-страницу. Код возврата 200 свидетельствует о том, что "все" прошло прекрасно. При этом сервер не забывает сказать "несколько слов о себе", идентифицируя себя с помощью строки Server: в данном случае у нас "работает" Web-сервер Netscape-Enterprise/3.5.1G.

Строка Content-Length означает, что броузеру было передано 2222 байта. На основе этих данных ваш броузер теперь сможет вычислить процент завершения загрузки страницы. Строка Content-Type определяет тип посланной обратно страницы. Для HTML-страниц указывается тип text/html, а для изображений может быть установлен тип image/jpeg.

По дате Last-Modified броузер может "судить" о том, была ли изменена страница с момента ее последней загрузки. Большинство Web-броузеров помещает загруженные ранее страницы в локальную *кэш-память,* чтобы при повторном обращении к этой странице не нужно было ее снова и снова загружать из Internet. При этом полученная от сервера дата сравнивается с датой сохраненной копии, находящейся в кэш-памяти. Если страница на сервере не была изменена, броузер использует локальную копию.

Пример: получение страницы вручную

При желании можно получить Web-страницу вручную. К этой возможности программисты часто прибегают при тестировании CGI-программ, чтобы убедиться в том, что Web-сервер посылает корректные ответы.

Для выполнения этого примера вам потребуется специальная программа, называемая *Telnet*-клиентом. Telnet-клиент — это программа доступа к удаленному компьютеру с помощью программы эмуляции терминала, исходное назначение которой — обеспечить удаленное подключение к рабочим станциям UNIX. Однако она часто используется для задач отладки протокола HTTP.

Если у вас установлена система UNIX, то в ее поставку обязательно должна входить утилита telnet. Если вы используете Microsoft Windows, то программа telnet автоматически инсталлируется при установке протокола TCP/IP. Чтобы запустить Telnet-клиент, просто используйте команду Выполнить из системного меню Пуск. Если же у вас не установлена программа telnet или вы работаете в системе Macintosh, попробуйте поискать Telnet-клиент в Internet и загрузить его на свой компьютер.

Подключение к Web-серверу с помощью telnet осуществляется следующим образом:

```
$ telnet www.webserver.com 80
```

где www.webserver.com — имя Web-сервера, а 80 — номер порта, к которому вы хотите подключиться (именно этот порт обычно используется Web-серверами для установки соединения по протоколу HTTP). Если ваша программа telnet имеет графический интерфейс, то, возможно, придется установить эти значения в специальном диалоговом окне.

После подключения Telnet-клиента вы можете не получить никакого символа приглашения на ввод или сообщения о факте подключения. Не беспокойтесь: это нормальная ситуация. Сервер HTTP ожидает, что клиент "заговорит" первым, поэтому от сервера и не ожидается никакого приглашения. В системе UNIX вы получите сообщение, которое может иметь следующий вид:

```
Trying www.webserver.com
Connected to www.webserver.com
Escape character is '^]'
```

Работая в других операционных системах (Windows или Macintosh), вы не увидите подобного сообщения.

Теперь нужно аккуратно и побыстрее ввести следующее:

```
GET http://www.webserver.com/ HTTP/1.0
```

После ввода этой строки нажмите клавишу <Enter> *дважды.* Web-сервер должен ответить обычным HTTP-заголовком и страницей верхнего уровня для данного Web-узла, а затем закрыть сеанс связи.

Пример: получение нетекстовой информации

Ваша CGI-программа не обязательно должна возвращать броузеру HTML-код. В действительности CGI-программа может отсылать броузеру все, что тот сможет принять и обработать.

Функция header в CGI-модуле информирует броузер о том, данные какого типа он будет получать. Для этого используется заголовок MIME Content-Type, который описывает содержимое данных, следующих за ним. В результате броузер сразу "узнает", что ему нужно делать с полученными данными.

По умолчанию функция header посылает броузеру заголовок Content-Type типа text/html. И броузер "понимает", что за заголовком следует содержимое, представляющее собой текст в формате HTML.

Предупредив броузер о типе получаемых данных, вы можете таким образом управлять способом обработки этих данных броузером. Данные могут выводиться в виде изображений, звука, передаваться дополнительному модулю броузера (browser plug-in) или внешней программе, запускаемой броузером.

Чтобы заставить функцию header послать нечто, отличное от обычного заголовка типа text/html, используйте ключ -type:

```
print header {-type => Тип_MIME);
```

Среди тех типов MIME, указываемых в заголовке Content-Type, которые обычно посылаются броузеру, чаще других встречаются text/plain (для текстов, не подлежащих интерпретации броузером), image/gif и image/jpeg (для GIF- и JPEG-изображений), а также application/*appname* (для данных, относящихся к конкретному приложению с именем *appname*). Есть еще специальный тип MIME заголовка Content-Type application/octet-steam, означающий передачу необработанных двоичных данных, которые броузер должен просто сохранить в файле.

Описанные выше типы данных пригодятся вам на случай, если вы захотите создать Web-узел, показывающий "изображение дня" или рекламу Web-страниц. Ежедневное изменение Web-страннии для отражения нового образа может превратиться в проблему. А если при этом вас не будет на месте, кто обновит "изображение дня"? Чтобы существенно облегчить себе жизнь, можно создать статическую HTML-страницу и написать CGI-программу на языке Perl, которая бы автоматически каждый день выводила новое изображение.

Для решения проблемы поместите следующий HTML-код в тело Web-страницы:

```
<Body>
Изображение дня:
<IMG SRC="/cgi-bin/daily_image.cgi">
</Body>
```

В приведенном фрагменте HTML-кода обратите внимание, что в дескрипторе <IMG> указана CGI-программа, а не GIF- или JPEG-изображение. Затем вам потребуется папка с изображениями, которых должно хватить хотя бы на месяц. Вы можете использовать любые изображения, главное, чтобы их файлы имели расширение .jpeg. К тому же заметьте, что эту программу можно легко адаптировать для изображений в формате GIF.

CGI-программа daily_image.cgi может иметь вид, представленный в листинге 20.1.

Листинг 20.1. Программа, обеспечивающая вывод "изображения дня"

```
#!/usr/bin/perl -w

use strict;
use CGI qw(:all);
my($imagedir, $day, @jpegs, $error);

$imagedir="/web/htdocs/pic_of_day";
```

```
8:  $error="/web/htdocs/images/error.jpg";
9:
10: sub display_image {
11:     my($image)=@_;
12:     open(IMAGE, "$image") || exit;
13:     binmode STDOUT; binmode IMAGE;
14:     print <IMAGE>;
15:     close(IMAGE);
16:     exit;
17: }
18:
19: print header(-type => 'image/jpeg');
20:
21: # Дни месяца, 1-28, 29, 30 или 31
22: $day=(localtime)[3];
23: $day=$day-1;   # Мы хотим использовать дни 0-27, и т.д...
24:
25: opendir(IMGDIR, $imagedir) || display_image($error);
26: @jpegs=sort grep(/\.jpg$/, readdir IMGDIR);
27: closedir(IMGDIR);
28:
29: my $image="$imagedir/$jpegs[$day]";
30: $image=$error if (not defined $jpegs[$day]);
31: display_image($image);
```

Проведем анализ программы.

- Строка 7. В этой строке задается каталог, в котором располагаются файлы изображений. Этот каталог можно заменить другим, соответствующим физическому расположению ваших файлов изображений.
- Строка 8. Как ни странно, но, поскольку эта CGI-программа не выдает никакого текста и поскольку HTML-страница, в которую она встроена, не отображает результат в виде текста, вы не можете просто выводить сообщения об ошибках. Если каталог `$imagedir` невозможно будет открыть, то будет отображен .jpg-файл, имя которого содержится в переменной Serror.
- *Строки 10—16.* Эта процедура выводит изображения в стандартный выходной поток, который будет направлен броузеру. На Windows-платформах поток STDOUT рассматривается как текстовый файл, значит, при выводе .jpg-файла в поток STDOUT изображение будет искажено. Поэтому, чтобы сделать дескрипторы STDOUT и IMAGE бинарными, используется функция binmode. Под управлением системы UNIX нет необходимости в использовании функции binmode, но ее присутствие не повредит. Обратите внимание на строку 12: если изображение не удается открыть, нет смысла в выводе сообщения об ошибке, поэтому просто выполняется выход из программы.
- *Строка 19.* Эта строка выводит стандартный HTML-заголовок, за исключением того, что в строке Content-Type вместо обычного типа text/html используется image/jpeg.
- *Строка 25.* Открывается каталог с изображениями для чтения. Если каталог с изображениями не открывается, то вызывается функция `display_image()`, которой через переменную Serror передается имя файла с изображением ошибки.

- *Строка 26.* Эта строка посложнее других, поэтому на ней стоит остановиться. Сначала содержимое каталога читается с помощью функции readdir. Затем из списка извлекаются имена файлов с расширением .jpg. Наконец, полученный список сортируется и присваивается переменной `@jpegs`.

Подробнее о вызове CGI-программ

До сих пор мы говорили о двух разных методах запуска CGI-программ. Первый и самый очевидный — просто поместить ее URL в гиперссылку или ввести его в поле Адрес окна броузера. Например, чтобы выполнить CGI-программу с именем `time.cgi`, используйте следующую строку:

```
<A HREF="http://server/cgi-bin/time.cgi ">Щелкните здесь</A>
```

После установки сеанса связи Web-сервер выполнит CGI-программу time.cgi и отправит броузеру результат ее работы, т.е. образ новой Web-страницы. Это пример запуска простейшей CGI-программы, подобной той, которую мы рассматривали на 17-м занятии, "Введение в CGI".

Другой способ запуска CGI-программы — включение ее имени в код HTML-формы. Например, следующая форма вызывает CGI-программу `process.cgi` после щелчка на кнопке подачи запроса Submit:

```
<FORM METHOD=GET ACTION="/cgi-bin/process.cgi">
<INPUT TYPE=TEXT NAME=STUFF><BR>
<INPUT TYPE=SUBMIT>
</FORM>
```

Этот метод вызова CGI-программы имеет дополнительное преимущество: он позволяет передавать CGI-программе параметры для обработки. Пожалуй, это свойство форм можно назвать одним из самых полезных.

Передача параметров CGI-программе

Полезность и гибкость CGI-программ существенно возрастают за счет передачи им дополнительных параметров прямо из гиперссылки. Например, можно создать гиперссылку в документе, которая запускает CGI-программу `foo.cgi` и передает ей два параметра X и Y, равных некоторым величинам. Для этого в дескрипторе <A HREF> нужно задать URL специального вида. Формат этого URL показан на рис. 20.1.

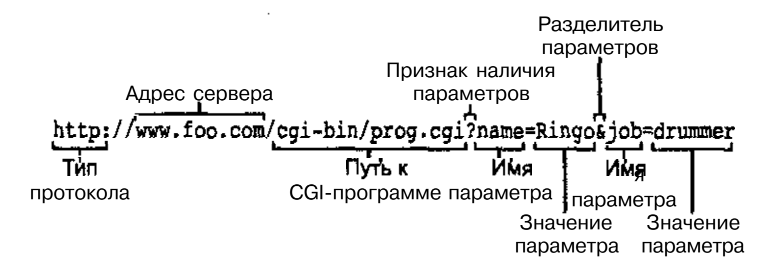


Рис. 20.1. URL с параметрами

Каждый параметр представляет собой имя некоторого значения, которое будет передано CGI-программе (подобно именованному элементу HTML-формы). Под значением параметра понимается некоторая строка, присваиваемая этому имени. Напри-

мер, чтобы создать гиперссылку, после щелчка на которой будет выполнена CGI-программа с параметрами sign и year, которым присвоены значения Овен и 1969 соответственно, используется следующий HTML-код:

```
<A HREF="http://www.server.com/cgi-bin/astrology.cgi?sign=Овен&
year=1969">Овен, год Петуха</A>
```

Внутри **CGI-программы** параметры обрабатываются функцией **param** модуля CGI обычным способом:

```
#!/user/bin/perl -w

use CGI qw{:all);
use strict;

print header;
print "Год ", param('year'), " и факт рождения под знаком ",
    param('sign'), " означает, что вы - выдающаяся личность.\n";
```

Передать можно любое количество параметров. Если нужно передать пустой параметр (без значения), просто используйте одно его имя, подобно тому, как мы поступили с параметром author в следующем примере:

```
<A HREF="http://www.server.com/cgi-bin/book.cgi?author=&
title=Beowolf">Beowolf</A>
```

Использование специальных символов

При вызове **CGI-программ** с параметрами следует знать о существовании некоторых специальных символов, которые не могут быть частью URL. Например, символ ? (знак вопроса) является специальным и отделяет основную часть URL от списка параметров. К числу других специальных символов относятся такие символы, как &, пробелы и кавычки.



Полный список специальных символов приведен в документе стандартов Internet RFC 2396.

Чтобы сделать один или несколько специальных символов частью URL, нужно закодировать их специальным образом. Это означает, что ASCII-значение этого символа следует преобразовать в двузначное шестнадцатеричное число, а затем поместить перед ним символ процента. Результат применения описанного "рецепта" кодирования для фразы "Привет, мир!" выглядит следующим образом[1]:

```
Привет%2C%20мир!
```

Вполне понятно, что в такой частично "зашифрованной" строке можно легко допустить ошибку и, вообще, ее вид малопривлекательный. Поэтому в модуле CGI предусмотрена функция, предназначенная для автоматического создания подобных строк. На следующем примере показано, как получить URL с нужной "шифровкой" специальных символов;

[1] *В этом примере русский текст использован только для наглядности. При передаче же кириллических символов в виде параметров URL их нужно закодировать точно так же, как и специальные символы.* — Прим. ред.

```
#!/user/bin/perl -w

use strict;
# Функция 'escape' должна быть указана явно
use CGI qv(:all escape);

print header;
my $string="Привет, мир!";
print '<A HREF="http://www.server.com/cgi-bin/parrot.cgi?message=',
      escape($string) , '">Щелкните здесь</A>';
```

Этот фрагмент программы создает правильно закодированную (с учетом требований к URL) HTML-ссылку. Обратите внимание на то, как в этом коде используется модуль CGI: use CGI qw(:all escape). При использовании модуля CGI функция escape обычно недоступна для программы, и в случае необходимости вам придется указать ее в операторе use в явном виде.

Следующая программа создает более длинный URL-адрес с закодированными значениями специальных символов:

```
#!/user/bin/perl -w

use strict;
use CGI qw{:all escape);

my %books=( Insomnia => 'S. King', Nutshell => 'O\'Reilly');
# Начинаем с основного URL
my $url="http://www.server.com/cgi-bin/add_books.cgi?";

# Выполняем сборку URL путем конкатенации "."
foreach my $title (keys %books) {
    $url .= escape($title); # Кодируем название и добавляем его
    $url .= "=";
    $url .= escape{$books{$title}); # Аналогично для автора
    $url.="&";
}

print header;
print "<A HREF=$url>Добавить книги в библиотеку</A>';
```

Построенное в результате этой программы значение URL будет иметь следующий вид:

```
http://www.server.com/cgi-
bin/add_books.cgi?Insomnia=S.%20King&Nutshell=O%27Reilly&
```

Последний символ & в конце URL игнорируется CGI-программой при выделении параметров с помощью функции param.

Включения на стороне Web-сервера

Часто при разработке Web-страницы оказывается, что большая часть ее содержимого абсолютно статична. Изредка меняются некоторые части Web-страницы, но в целом она остается такой же. Рассмотрим Web-страницу, на которой отображается курс акций некоторой компании. Подавляющая часть страницы будет, по всей вероятности, статичной — средства перемещения, изображения, эмблемы, информация об

использовании, верхние и нижние колонтитулы, заголовки и пр. Самая важная часть этой страницы — курс акций — генерируется путем считывания нужной информации из базы данных и заполнения соответствующих полей.

Чтобы облегчить создание подобных страниц, большинством Web-серверов поддерживается средство, называемое *включениями на стороне сервера* (server-side includes — SSI), которое также называется *средством синтаксического анализа* сервера (server-parsed HTML). Это средство позволяет создателю Web-узла на базе языка HTML строить Web-страницу, которая по большей части статична, но при этом с помощью Web-сервера изменять отдельные ее части (рис. 20.2). Такую Web-страницу можно представить в виде бланка, который нужно заполнить, а заполнение этого "бланка" выполняется CGI-программой.

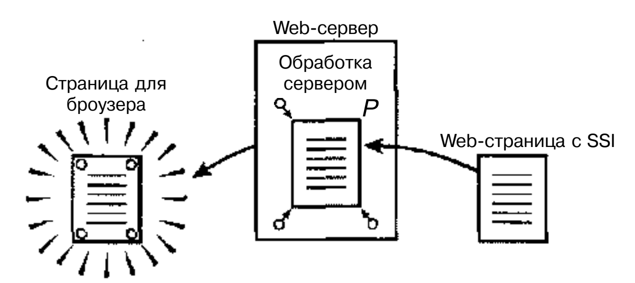


Рис. 20.2. Web-сервер может добавлять некоторые элементы к HTML-странице



Администратор вашего Web-сервера должен активизировать работу SSI. Чтобы сервер должным образом "понимал" HTML-код с встроенными SSI-элементами, вам, возможно, придется присваивать таким HTML-страницам специальные расширения `.shtml` или `.stm`. Обратитесь к документации по Web-серверу или к администратору узла, чтобы выяснить, как используется средство SSI на данном конкретном сервере, поскольку директивы и синтаксис могут различаться от сервера к серверу.

Web-сервер во время чтения статической HTML-страницы с диска отыскивает дескрипторы, которые заставляют его выполнять подстановку значений. Например, дескриптор `<!--#echo var="LAST_MODIFIED"-->` заставит Web-сервер Apache подставить вместо него дату последней модификации текущей HTML-страницы. Броузер не "видит", как происходит этот процесс. Он видит только дату, уже подставленную сервером. Проиллюстрируем процесс подстановки на следующем примере.

Исходная Web-страница	Преобразованная Web-страница
`<HTML>`	`<HTML>`
`<BODY>`	`<BODY>`
Дата последней модификации:	Дата последней модификации:
`<!--#echo var="LAST_MODIFIED"-->`	Wednesday, 01-Jul-2000 21:29:31 EDT
`</BODY>`	`</BODY>`
`</HTML>`	`</HTML>`



Средства SSI у разных Web-серверов реализованы по-своему. Поэтому дескрипторы подстановки часто имеют различный синтаксис, а отдельные их виды поддерживаются далеко не всеми серверами. А некоторые Web-серверы вообще не поддерживают средств SSI, например Personal Web Server компании Microsoft. Дескрипторы SSI, используемые на этом занятии, совместимы с такими самыми популярными Web-серверами (на момент написания этой книги), как Apache и Internet Information Server компании Microsoft.

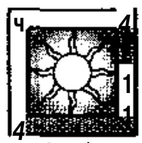
Цель этого занятия состоит отнюдь не в том, чтобы вы узнали все о средствах SSI, поскольку их слишком много и к тому же они зависят от типа Web-сервера. Тем не менее вам следует познакомиться с дескриптором SSI **#exec**, который в HTML-файле имеет следующий вид:

```
<!--#exec cgi="/cgi-bin/stockprice.cgi"-->
```

Встретив дескриптор **#exec**, Web-сервер запустит **CGI-программу** stockprice.cgi. Результат работы этой **CGI-программы** помешается в HTML-**поток**, который направляется броузеру. После завершения CGI-программы остальная часть HTML-файла (расположенная после дескриптора **#exec**) посылается броузеру как обычно.

Пример: работа с дескрипторами SSI

Для данного примера создадим простую страницу, которая выводит текст "Привет, мир!", и картинку, изменяющуюся в зависимости от времени суток. Для начала нам потребуются два рисунка: один — для ночи, другой — для дня (рис. 20.3),




day.jpg night.jpg

Рис. 20.3. Изображения дня и ночи: day.jpg и night.jpg

Затем нам потребуется HTML-файл с приветственным сообщением (см. ниже). Если вы создаете этот пример для себя, не забудьте о том, что ему, скорее всего, придется назначить расширение `.shtml` или `.stm`, чтобы Web-сервер смог распознать дескрипторы SSI.

```
<HTML>
<HEAD>
<TITLE>Страница приветствия</TITLE>
</HEAD>
<BODY>
Добро пожаловать на мою Web-страницу. Сейчас из окна я вижу:
<!--#exec cgi="/cgi-bin/sunmoon.cgi"-->
</BODY>
</HTML>
```

Текст программы `sunmoon.cgi` приведен в листинге 20.2.

Листинг 20.2. Программа приветствия, выводящая изображения дня и ночи

```
1:  #!/usr/bin/perl -w
2:
3:  use CGI qw(:all);
4:
5:  # Узнаем время в 24-часовом формате с помощью функции localtime().
6:  my $hour=(localtime)[2];
7:  my $image;
8:
9:  # Считаем, что до 6 утра и после 6 вечера - на дворе ночь
10: if ($hour<6 or $hour>18) {
11:         $image="night.jpg";
12: } else {
13:         $image="day.jpg";
14: }
15: print header;
16: print qq{<IMG SRC="$image" ALT="$image">\n};
```

АНАЛИЗ Проведем анализ программы.

- *Строка 3.* Поскольку это CGI-программа, необходимо подключить модуль CGI. Директива `qw(:all)` гарантирует возможность использования любых функций модуля.
- *Строка 6.* Функция localtime в контексте списка возвращает список элементов, описывающих текущее время (подробнее об этом мы говорили на 4-м занятии, "Укладка строительных блоков: списки и массивы"). При заключении в круглые скобки функция localtime рассматривается в контексте списка, а элемент `[2]` означает, что будет возвращен третий элемент списка (счет ведется с нуля) и присвоен переменной `$hour`. Этот элемент списка представляет собой время в 24-часовом формате.
- *Строка 15.* Заголовок, оговоренный протоколом HTTP, по-прежнему должен выводиться с помощью функции header модуля CGI, несмотря на то, что, казалось бы, половина Web-страницы уже готова.
- *Строка 16.* В дескрипторе <IMG>, предназначенном для вывода изображения, указывается либо значение $day, либо значение $night, а дескриптор ALT используется на случай, если броузер не сможет вывести это изображение.

Текст Web-страницы, отображенной броузером в 8 часов утра, будет иметь следующий вид:

```
<HTML>
<HEAD>
<TITLE>Страница приветствия</TITLE>
</HEAD>
<BODY>
Добро пожаловать на мою Web-страницу. Сейчас из окна я вижу:
<IMG SRC="day.jpg" ALT="day.jpg">
</BODY>
</HTML>
```

Выглянем из окна

БОЛЬШИНСТВО рассмотренных до сих пор функций из модуля CGI предназначены либо для управления броузером (функция header), либо для обработки параметров, переданных **CGI-программе** (функции escape и **param**). Кроме них в модуле CGI разработан целый набор функций, предназначенных для получения информации о типе броузера и сервера. Получить представление о таких функциях можно из табл. 20.1, а полный их список приведен в электронной документации на модуль CGI, к которой можно получить доступ, введя после приглашения команду perldoc CGI.



Большинство описанных в этом разделе функций зависит от значений, которые предоставляются Web-сервером или посылаются Web-броузером в сеансе связи с сервером через протокол **HTTP**. При этом следует иметь в виду, что Web-броузеры не всегда сообщают точные сведения (например, что касается значения referer или user_agent), да и **Web-серверы** могут обмануть ваши ожидания (например, в отношении значения server_name).

Таблица 20.1. Неполный список информационных функций

Функция	Описание
referer	Возвращает URL гиперссылки, с помощью которой будет осуществлен переход к этой странице. (Да, сформулировано неправильно. В оригинальном документе стандартов Internet, описывающем это поле, присутствовала орфографическая ошибка, которая намеренно не исправлена для согласованности документов.)
user_agent	Возвращает строку, идентифицирующую тип Web-броузера, который запросил страницу (например, броузеры Netscape, IE или Lynx)
remote_host	Возвращает либо имя компьютера (host), **либо** IP-адрес системы, которая запросила страницу. Получаемое значение зависит от того, как сконфигурирован ваш Web-сервер и доступно ли для него имя удаленного компьютера (hostname)
script_name	Возвращает имя данной CGI-программы в виде части URL (например, **/cgi-bin/foo.cgi**)
server_name	Возвращает имя сервера, на котором работает **CGI-программа**
virtual_host	Возвращает имя виртуального Web-сервера, который использовался для выполнения данной CGI-программы. Эта функция отличается от функции **server_name**, поскольку зачастую один сервер может управлять несколькими Web-узлами. **Функция virtual_host** возвращает имя конкретного узла, который был затребован

Использование этих функций демонстрируется на примере следующей короткой программы:

```
#!/usr/bin/perl -w

use strict;
use CGI qw(:all);

print header;
```

```
print "Запрошено из: ", referer, "<BR>";
print "Вы работаете с броузером: ",
             user_agent, "<BR>";
print "Имя компьютера: ", remote_host,
             "<BR>";
print "Имя данной программы: ",
             script_name, "<BR>";
print "Программа выполняется на сервере: ",
             server_name, "<BR>";
print "Имя виртуального сервера: ",
             virtual_host, "<BR>";
```

При выполнении этой программы на тестовом Web-сервере были получены следующие результаты:

```
Запрошено из: http://testsys.net/links.html
Вы работаете с броузером: Mozilla/4.51 [en] (Win95; I)
Имя компьютера: 192.168.1.2
Имя данной программы: /cgi/showstuff.cgi
Программа выполняется на сервере: testsys
Имя виртуального сервера: perlbook
```

Перенаправление

Одним из полезных приемов, применяемых в CGI-программах, является *перенаправление,* или *переадресация* HTTP-запроса (redirection). Перенаправление используется в случаях, когда нужно, чтобы CGI-программа загрузила другую страницу на основе некоторого вычисляемого значения.

Предположим, что вы создали набор страниц, предназначенных для просмотра броузером заданного типа. И хотя такую практику нельзя назвать удачной, тем не менее бывают случаи, когда страницы содержат элементы, обрабатываемые только дополнительными модулями (plug-in) броузера Netscape для Windows. В таком случае следует создать несколько копий Web-узла, предназначенных для просмотра разными броузерами. Для обработки входящих запросов и перенаправления их по корректному URL придется написать небольшую CGI-программу (рис. 20.4).



Рис. 20.4. Схема процесса перенаправления запроса с одного Web-узла на другой

Для реализации перенаправления необходимо использовать функцию модуля CGI redirect, которая управляет рассмотренным выше процессом HTTP-запроса и заставляет броузер загрузить новую страницу.

В листинге 20.3 содержится короткая программа, предназначенная для перенаправления пользователей броузера Netscape под управлением Windows к одной странице, а всех остальных — к другой.

Листинг 20.3. Перенаправление в зависимости от типа броузера

```
1:  #!/usr/bin/perl -w
2:
3:  use CGI qw(:all);
4:  use strict;
5:  my($browser, $target);
6:
7:  # Определение типа броузера
8:
9:  $browser=user_agent;
10: $target="http://www.server.com/generic.html";
11:
12: # Проверка для WinXX и Netscape
13: if ($browser=~/Mozilla/ and $browser=~/Win/) {
14:     $target="http://www.server.com/netscape.html";
15: }
16: print redirect( -uri => $target );
```



Проведем анализ программы.

- *Строка 9.* Переменной $browser присваивается тип броузера.
- *Строка 10.* Стандартный URL запоминается в переменной $target. Все броузеры, отличные от Netscape, будут отосланы по этому адресу.
- *Строки 13—14.* Идентификационные данные броузера, сохраненные в переменной `$browser`, проверяются на наличие в них слов Mozilla и Win, и при положительном результате назначается новый адрес.
- *Строка 16.* Броузеру посылается сообщение о перенаправлении.

Перенаправление посредством CGI работает всегда, в то время как при использовании других методов (например, JavaScript- и HTML-расширений) возникают проблемы. JavaScript не поддерживается всеми платформами, а присваивание свойству `window.location.href` в JavaScript нового URL может не дать желаемых результатов. Использование для перенаправления дескриптора HTML `<META HTTP-EQUIV="refresh">` приведет к существенной отсрочке выполнения самого процесса перенаправления, поскольку броузеру перед этим придется полностью загрузить указанную страницу. JavaScript также "грешит" этой проблемой. А перенаправление через протокол HTTP с помощью CGI-сценария выполняется до того, как будет передан какой бы то ни было HTML-код броузеру и притом практически мгновенно.



> Броузер Netscape при использовании функции user_agent модуля CGI идентифицирует себя как Mozilla. Это имя происходит от имени оригинального графического Web-броузера *Mosaic.* Имя, возвращаемое популярным броузером Windows 95 Netscape 4.51 при использовании функции user_agent, выглядит как `Mozilla/4.51 - (Win95; I)`.

Резюме

На этом занятии вы узнали о том, что происходит, когда Web-страница запрашивается с сервера, и получили некоторые сведения о протоколе **HTTP.** Кроме того, вы научились вызывать из гиперссылки CGI-программу и передавать ей параметры, использовать средства SSI сервера. Вы также узнали, как управлять процессом перенаправления броузера с помощью протокола HTTP и получить информацию о броузере и сервере.

Вопросы и ответы

Создается впечатление, что примеры, в которых используются средства SSI, не работают.

Средства SSI могут не работать по нескольким причинам. Во-первых, вам следует убедиться в том, что ваш Web-сервер поддерживает SSI. Во-вторых, удостоверьтесь, что средствам SSI на вашем Web-сервере дан "зеленый свет", т.е. они не запрещены для использования. В-третьих, убедитесь, что ваши HTML-файлы имеют нужные расширения, которые позволят им быть воспринятыми средствами SSI. Возможно, чтобы выяснить эту информацию, вам стоит обратиться к администратору сервера. Наконец, убедитесь, что вы используете правильный синтаксис для дескрипторов HTML, предназначенных для работы с SSI.

Если вы используете дескриптор <!~#exec cgi-->, убедитесь, что ваша CGI-программа нормально работает и при обычном запуске (без средств SSI).

С помощью режима просмотра исходного кода Web-страницы, который есть практически в любом броузере, после загрузки им какой-нибудь страницы можно легко определить, поддерживаются ли средства SSI на этом сервере. Если в исходном HTML-коде вы увидите дескрипторы SS1, значит, этот сервер их не распознал и не проанализировал.

Что делать, если не работает пример с использованием программы Telnet-клиента?

Если программа telnet не выполняет соединение, убедитесь, что вы ввели правильный адрес Web-сервера и указали корректный номер порта, по всей вероятности 80. Чтобы убедиться в правильности номера порта, стоит заглянуть в документацию на программу telnet.

Еще одной распространенной проблемой является невозможность просмотреть результат собственного ввода символов. Некоторые Telnet-клиенты отображают их для вас, а некоторые нет. Не стоит беспокоиться по этому поводу: старайтесь просто аккуратно вводить символы. После ввода строки GET не забудьте *дважды* нажать клавишу <Enter>.

Семинар

Контрольные вопросы

1. Будет ли следующий URL работать ожидаемым образом?

   ```
   <A HREF="/cgi/foo.pl?name=Ben Franklin&Job=printer">
   ```

   а) да;

   б) нет, вы не можете *так* передать два параметра в CGI-программу;

   в) нет, пробел в имени Ben Franklin не разрешено использовать в таком виде.

2. Что обеспечивает обработку включений на стороне сервера (server-side includes)?

   а) броузер;

   б) Web-сервер;

   в) операционная система.

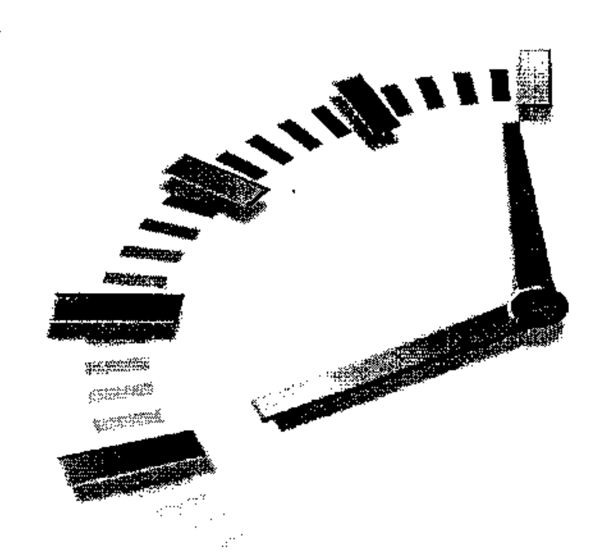
Ответы

1. Правильный ответ — вариант в). Необходимо использовать кодирование специальных символов для "сокрытия" пробела.

2. Правильным будет вариант б). Web-сервер преобразует дескрипторы SSI (в HTML-исполнении) в конечные значения, прежде чем посылать их броузеру.

Упражнения

- Попытайтесь с помощью программы telnet подключиться к одному из ваших любимых Web-узлов и вручную получить от сервера информацию.



21-й час

Файлы cookie

Из материала 19-го занятия, "Сложные формы", вы узнали о том, как с помощью скрытых полей в HTML-форме можно заставить свой Web-броузер "запомнить" содержимое предыдущих Web-страниц. Этот процесс необходимо хорошо понимать, поскольку время от времени вам придется передавать информацию из одного экземпляра CGI-программы другому, и единственный путь решения этой задачи — сохранить нужные данные с помощью броузера.

Помимо способа хранения информации, рассмотренного на предыдущем занятии, существует еще один, в котором также использованы возможности броузера. Речь идет о так называемых файлах *cookie,* которые представляют собой данные, передаваемые между броузером и CGI-программой во время сеанса *HTTP-связи.* В cookie-методе сохранения информации с помощью броузера заложено намного больше гибкости по сравнению с методом использования скрытых полей.

Основные темы этого занятия.

- Что такое файлы cookie.
- Как осуществить обработку файлов cookie.
- Как избежать проблем, связанных с файлами cookie.

Что такое файлы cookie

Файлы cookie можно представить себе в виде билета в кинотеатр. Вы можете пойти в кассу кинотеатра и купить билет на любой удобный для вас сеанс. После этого вы вольны уйти, купить пакет воздушной кукурузы и заняться своими делами. Когда подойдет время сеанса, вам нужно предъявить билет контролеру, которого совершенно не интересует, как, когда или почему вы приобрели этот билет, но коль он у вас есть, контролер пропустит вас в кинозал. Билет дает право его предъявителю посмотреть кинофильм на более позднем (по сравнению с моментом покупки билета) сеансе.

Файлы cookie — это просто пакет информации, который CGI-программа "просит" сохранить броузер. Этот пакет броузер может в любое время переслать обратно этой или же другой CGI-программе. Файлы cookie передаются также обратно серверу при

запросе обычных HMTL-страниц. Пакет cookie может содержать информацию любого вида: о многостраничных Web-формах, данные о посещаемости Web-страниц, предпочтениях пользователей и т.д.

Пакет cookie передается от сервера броузеру в случае, если CGI-программа запрашивает создание cookie (рис. 21.1), и этот процесс называется *установкой cookie.*

Пакет cookie может быть впоследствии использован CGI-программой, которой броузер отправляет информацию, сохраненную в этом пакете, как показано на рис. 21.2.

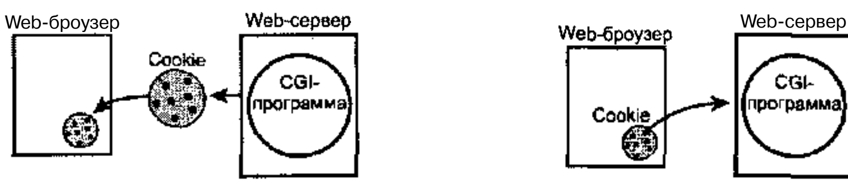


Рис. 21.1. Передача пакета cookie броузеру из CGI-программы

Рис. 21.2. Броузер возвращает пакет cookie серверу

Откуда такое странное название — cookie?

В компьютерных кругах *cookie* — очень старый термин. Он относится к любому биту информации (передаваемому программами или подпрограммами), который позволяет владельцу cookie выполнить определенную операцию. Некоторые виды пакетов cookie называются *magic cookies* (магические пакеты cookie), поскольку они содержат данные, имеющие смысл только для их отправителя и получателя. CGI-cookie не относятся к числу магических.

Создание пакетов cookie

Для создания пакета cookie можно использовать функцию модуля CGI под именем cookie. Вот ее синтаксис:

```
$cookie_object=cookie( -name => cookie_имя,              # необязательный
                       -value => cookie_значение,
                       -expires => дата_окончания,       # необязательный
                       -path => информация_о_пути,       # необязательный
                       -domain => информация_о_домене,   \ необязательный
                       -secure => true/false             # необязательный
);
```

Аргументы функции cookie передаются необычным способом. Каждый аргумент в обращении к функции cookie является именованным. Такой способ передачи аргументов в языке Perl очень удобен, поскольку не нужно помнить порядок следования аргументов: достаточно при использовании указать его имя.

После вызова с использованием этого синтаксиса функция cookie возвращает ссылку на объект типа cookie (ее нужно сохранить в скалярной переменной). Эта ссылка передается функции header модуля CGI, которая и отправляет пакет cookie броузеру. Единственным обязательным аргументом для создания файла cookie является аргумент -value. С помощью аргумента -name броузеру можно послать несколько пакетов cookie одновременно, при этом их выборка может быть как групповой, так и индивидуальной. Другие аргументы——expires, -path, -domain и -secure — будут рассмотрены в следующем разделе.

Функция header в модуле CGI выполняет передачу пакета cookie броузеру. Это значит, что вы должны создать пакет, используя функцию cookie, а затем вызвать функцию header. Не следует посылать броузеру данные какого-либо другого типа до тех пор, пока не будут посланы файл cookie и заголовок.

Чтобы создать файл cookie и отослать его броузеру, можно использовать CGI-программу, текст которой приведен ниже:

```
#!/usr/bin/perl -w
use CGI qw(:all);
use strict;

my $cookie=cookie(-name => 'Sample',
    -value => 'Этот cookie не включает сообщений');

# Передаем cookie броузеру
print header(-cookie => $cookie};
```

После выполнения приведенного фрагмента программы файл cookie с именем Sample будет установлен в броузере. Новоиспеченный файл cookie будет содержать информацию Этот cookie не включает сообщений.



> В действительности не исключено, что cookie окажется не установленным. Броузеры могут отказаться принять cookie, и для этого есть немало причин (см. раздел "Проблемы с файлами cookie" этого занятия).

Для считывания пакетов cookie из броузера в CGI-программу используется та же функция cookie. Вызванная без аргументов (как показано в следующих примерах), она возвращает список имен всех файлов cookie, которые броузер передал вашему серверу:

```
@cookie_list=cookie(); # Возвращает имена всех файлов cookie

--или--

# Возвращает значение для конкретного файла cookie
$cookie_value=cookie{$cookie_name);
```

По умолчанию после установки в броузере файл cookie возвращается в любую CGI-программу, которая размещается на том же самом сервере, т.е. только тот сервер, который установил пакет cookie, может осуществлять их выборку. Чтобы просмотреть содержимое созданного выше файла cookie Sample, можно использовать другую CGI-программу:

```
#!/usr/bin/perl -wT
use CGI qw(:all);
use strict;

print header(); # Выводит стандартный заголовок

print "Значение cookie 'Sample': ", cookie('Sample'), "<P>";
```

В предыдущем фрагменте кода функция cookie была вызвана с одним аргументом — именем файла cookie, который нас интересует. Это значение считывается и выводится для просмотра.

Пакет cookie должен быть возвращен броузером до завершения его работы. После повторного запуска от нашего cookie Sample не останется и следа. Для создания более "устойчивых" пакетов cookie обратитесь к разделу "Сохранение файлов cookie" этого занятия.



> Большинство броузеров обладают возможностью оповещения факта установки файлов cookie. В броузере Netscape соответствующие опции можно найти в разделе Preferences вкладки Advanced. А в броузере Internet Explorer вам придется для этого открыть вкладку Дополнительно (Advanced) диалогового окна Свойства обозревателя (Internet Options) и отыскать переключатель, управляющий режимом установки файлов cookie.

Пример: использование файлов cookie

В качестве примера создадим небольшую программу, с помощью которой можно будет менять в Web-броузере цвета просматриваемой в данный момент страницы. На самом деле эта программа выполняет сразу несколько функций, как описано ниже.

1. Проверяет наличие изменения в цвете фона, установленного по умолчанию, путем проверки параметров программы.
2. Устанавливает файл cookie в броузере с использованием нужного цвета фона.
3. Устанавливает цвет фона страницы в соответствии с нужным цветом.
4. Отображает CGI-форму, позволяющую выбрать цвет.

Программа изменения цвета Web-страницы представлена в листинге 21.1.

Листинг 21.1. Полный вариант программы ColorChanger

```
#!/usr/bin/perl -w
use strict;
use CGI qw(:all);
use CGI::Carp qw(fatalsToBrowser);
my($requested_color, $old_color, $color_cookie)=("","");
 $old_color="blue";  # Стандартный цвет
# Был ли запрос на установку нового цвета?
if (defined param('color')) {
        $requested_color=param('color');
}
# Каким был старый цвет?
if (defined cookie('bgcolor')) {
        $old_color=cookie('bgcolor');
}
if ($requested_color and ($old_color ne $requested_color)) {
        # Установка cookie в броузере
        $color_cookie=cookie(-name => 'bgcolor',
                             -value => $requested_color);
        print header(-cookie => $color_cookie);
} else {
        # Ничего не изменено, нет нужды в установке cookie
        $requested_color=$old_color;
        print header;
}
print<<END_OF_HTML;
<HTML>
<HEAD>
<TITLE>Установка цвета фона</TITLE>
</HEAD>
<BODY BGCOLOR="$requested_color">
<FORM>
```

```
32: <SELECT NAME="color">
33:        <OPTION value='red'>Красный
34:        <OPTION value='blue'>Синий
35:        <OPTION value='yellow'>Желтый
36:        <OPTION value='white'  >Белый
37: </SELECT>
38: <INPUT TYPE=SUBMIT VALUE="Установка цвета">
39: </FORM>
40: </BODY>
41: </HTML>
42: END OF HTML
```


Проведем анализ программы.

- *Строки 7—10.* Если эта CGI-программа вызывается для обработки данных HTML-формы, функция param('color') возвращает значение выбранного цвета. В противном случае значение переменной $requested_color остается неопределенным.
- *Строки 12—14.* В этих строках программы проверяется факт существования файла cookie с именем bgcolor (его ведь может и не быть!). Если файл действительно существует, его содержимое (значение цвета фона экрана, сохраненное в файле cookie в последний раз) запоминается в переменной $old_color.
- *Строки 15—19.* Если цвет изменен (содержимое файла cookie не совпадает с новым значением), нужно установить в броузере новый файл cookie с обновленным значением цвета.
- *Строки20—24.* В противном случае выводится только заголовок, без cookie. Учтите, что броузер будет сохранять предыдущий файл cookie неограниченно долго.
- *Строки 25—42.* Здесь создается обычная форма HTML. Однако обратите внимание на строку 30 — именно в ней выполняется изменение цвета HTML-страницы.

Еще один пример: просмотр файлов cookie

Удивительно короткую программу просмотра файлов cookie, приведенную в листинге 21.2, можно рассматривать как вспомогательное средство при отладке CGI-программ, работающих с пакетами cookie. Эта программа выводит список всех пакетов cookie, хранимых в Web-броузере, которые считаны с одного и того же Web-сервера.

Листинг 21.2. Программа просмотра файлов cookie

```
#!/usr/bin/perl -w

use strict;
use CGI qw(:all);

print header();

print "Список файлов cookie, которые можно просмотреть:<P>";

foreach my $cookie (cookie()) {
        print "Имя cookie: $cookie <BR>";
        print qq{Значение cookie: "}, cookie($cookie), qq{"<P><HR>};
}
```

Проведем анализ программы.

- *Строка 10.* Имена всех пакетов cookie считываются с помощью функции cookie и по очереди присваиваются переменной `$cookie`.
- *Строки 11–12.* Выводится имя и значение каждого пакета cookie.

Работа этой программы просмотра основана на получении списка всех доступных пакетов cookie с помощью функции `cookie()`, получении значения каждого пакета и распечатке его вместе с именем cookie.

Более сложные методы работы с файлами cookie

Основы работы с файлами cookie довольно просты. Сервер передает броузеру пакет cookie, а позже броузер возвращает его назад серверу. Хотя, конечно, существуют нюансы. Например, можно увеличить время жизни (или срок действия) файлов cookie, а также сделать его бесконечно большим. Такие файлы cookie называются *перманентными.* Их параметры можно задать так, чтобы они возвращались только по конкретному URL. (Данная настройка означает применение определенного уровня безопасности.)

Сохранение файлов cookie

Все рассматриваемые нами до сих пор файлы cookie временно хранились в броузере: как только броузер завершал свою работу, файлы cookie автоматически удалялись. Использование временных файлов cookie вместо скрытых полей HTML-форм для передачи данных между несколькими формами вполне приемлемо. При новом запуске броузера вы вряд ли захотите использовать старый файл cookie, оставшийся от предыдущего сеанса работы с сервером, поскольку пользователь должен будет заново заполнить все поля многостраничной формы.

В некоторых случаях возникает необходимость в постоянном хранении файлов cookie (речь идет о нескольких дня, неделях или даже месяцах). С помощью модуля CGi в среде Perl создание таких файлов не составляет труда.

Чтобы установить дату истечения срока действия файла cookie, при его создании нужно использовать аргумент -expires. За ним должна следовать дата, определяющая конец срока действия файла cookie. Эту дату можно указать в нескольких форматах, как показано в табл. 21.1.

Таблица 21.1. Форматы даты истечения срока действия файлов cookie

Формат	Пример	Описание
Количество секунд	+30s	Через 30 секунд от данного момента
Количество минут	+15га	Через 15 минут от данного момента
Количество часов	+12h	Через 12 часов от данного момента
Количество месяцев	+6M	Через 6 месяцев от данного момента
Количество лет	+1y	Через 1 год от данного момента

Формат	Пример	Описание
Сиюминутно	now	Срок действия cookie истекает немедленно
Любое отрицательное время	-10m	Срок действия cookie истекает немедленно
Точное время	Saturday, 28-Aug-2000 22:51:05 EEST	

Окончание табл. 21.1

При указании точного времени вы должны использовать формат, приведенный в табл. 21.1. Все другие возможные величины представляют собой интервал времени, отсчитываемый от текущего времени. При их использовании нужное время (в "полноформатном" виде) будет вычислено без вашего участия и отослано броузеру.

Следующая небольшая программа устанавливает в броузере файл cookie, срок действия которого истекает через восемь дней:

```
#!/usr/bin/perl -w
use CGI qw(:all);
use strict;

my $cookie=cookie(-name => 'Favorite',
    -value => 'Мягкое овсяное печенье с изюмом',
     -expires => '+8d' );

# Передаем cookie броузеру
print header(-cookie => $cookie);
```

А теперь поговорим немного о грустном

В мире нет ничего постоянного. В том числе и постоянно действующих файлов cookie. Например, если ваша CGI-программа отправит броузеру некоторый пакет cookie, не следует полагать, что он будет активен в течение заданного (с момента установки) количества недель, месяцев или лет.

Как будет отмечено ниже, в разделе "Проблемы с файлами cookie" броузеры не обязаны хранить файлы cookie. На самом деле они вообще не обязаны их принимать, причем вы даже не будете уведомлены о том, что файл cookie не принят.

Броузеры в любой момент могут избавиться от файлов cookie, чтобы освободить место для новых, полученных от других серверов, или вообще удалить их без всяких причин. Некоторые броузеры могут разрешить пользователям редактировать файлы cookie или создавать новые экземпляры.

Пользователи могут случайно или намеренно удалить файлы cookie. Если пользователь устанавливает новую версию броузера или операционной системы, файлы cookie могут быть попросту затерты или оказаться "не в том месте". Иногда файлы cookie исчезают даже при запуске другого броузера. Когда броузер неактивен, пакеты cookie обычно хранятся в файле, который пользователи могут легко отредактировать, удалить или испортить.

Следовательно, хранение ценной информации в файлах cookie нельзя назвать удачной идеей. Любая информация, которую, казалось бы, стоило постоянно сохранять в пакете cookie, может быть легко заменена, удалена или испорчена. Речь идет о пользовательских параметрах настройки, именах и паролях, предназначенных для регистрации на Web-сервере, различного рода служебной информации, времени последнего посещения и др.



Если хотите знать, то большинство броузеров, находясь в неактивном состоянии, хранят пакеты cookie в обычных текстовых файлах, поэтому их можно легко просмотреть с помощью любого текстового редактора. Броузер Netscape хранит пакеты cookie в файле `cookies.txt`, расположенном в рабочем каталоге пользователя, который в разных операционных системах имеет разное имя. Броузер Internet Explorer хранит файлы cookie в каталоге `\Windows\Cookies`.

Отправка файлов cookie другим серверам

Файлы cookie по умолчанию отправляются обратно только тому серверу, который их прислал. Иногда такая оправка пакетов cookie "по обратному адресу" отвечает вашему желанию, а иногда — совсем нет. Например, рассмотрим некоторый вымышленный Web-сервер congo.com, который предназначен для продажи книг. На нем работают два виртуальных Web-узла: `www.congo.com` и shopping.congo.com (рис. 21.3). Основной Web-узел (с адресом `www.congo.com`) содержит всю информацию о компании, гиперссылки на другие узлы и, что более важно, гиперссылки на электронный книжный магазин.

Web-узел `www.congo.com` содержит HTML-форму для регистрации и обрабатышающую ее данные CGI-программу. С их помощью пользователь может внести свое имя в списки почтовой рассылки и сообщить серверу о том, какие книги его интересуют. Позже, при посещении узла www.congo.com, пользователь может прочитать о новых книгах, список которых составляется с учетом его интересов. Эта информация хранится в файлах cookie, полученных с узла www.congo.com и установленных в броузере пользователя (рис. 21.4).

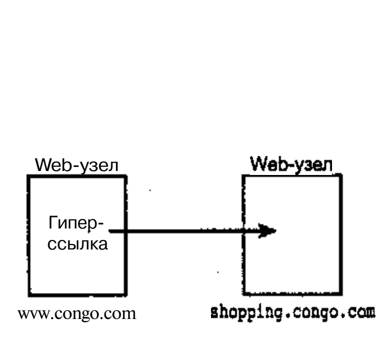


Рис. 21.3. Два виртуальных Web-узла, связанных вместе

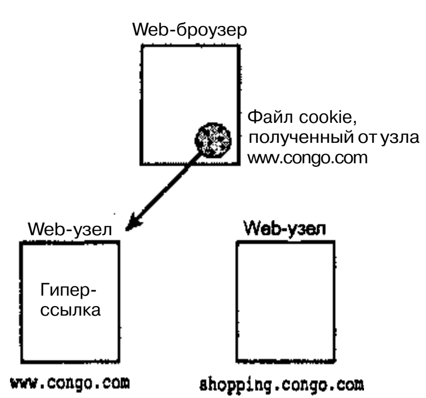


Рис. 21.4. По умолчанию пакеты cookie возвращаются только тому виртуальному Web-узлу, от которого они получены

Проблема состоит в том, что, когда пользователь от узла www.congo.com переходит к электронному магазину по адресу shopping.congo.com, файлы cookie не отправляются серверу с адресом shopping.congo.com. Файлы cookie возвращаются только тому серверу, от которого они были получены. Если пакет cookie был послан с адреса www.congo.com, то он не возвратится по адресу shopping.congo.com.

Так что же делать? Совершенно неприемлемо заставлять пользователя заполнять другую форму и отправлять ему новый cookie с сервера shopping.congo.com. Есть выход получше. Можно ограничить действие файла cookie конкретным именем домена. Например, когда с сервера www.congo.com посылается оригинальный пакет cookie, можно создать условия, при которых этот пакет мог бы отправиться к любому Web-узлу домена congo.com, как показано на рис. 21.5.

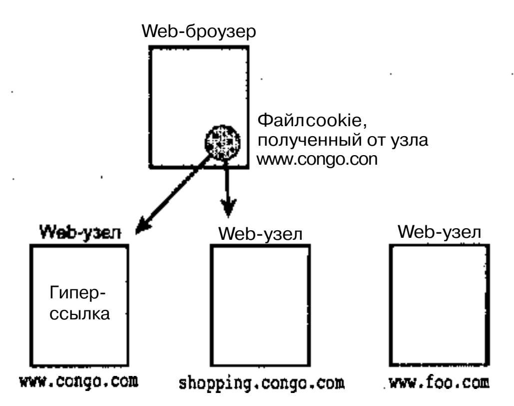


Рис. 21.5. Файлы cookie, возвращаемые к обоим Web-узлам

Эта технология реализуется на этапе создания пакета cookie с помощью аргумента -domain функции cookie:

```
$cookie=cookie( -name => 'preferences',
        -value => 'mysteries, horror',
        -domain => 'congo.com'};
print header(-cookie => $cookie);
```

В приведенном выше фрагменте создается пакет cookie с именем preferences, действие которого ограничивается доменом congo.com. Теперь броузер сможет возвратить этот cookie любому Web-серверу, имя которого оканчивается на congo.com.



Аргумент для задания домена должен иметь по крайней мере две части. Он не может являться доменом верхнего уровня, т.е. представлять собой имена `.com` или `.net`. Это сделано для того, чтобы броузер не передавал один и тот же пакет cookie всем серверам домена верхнего уровня.

Создание персональных пакетов cookie

Можно также ограничить область действия пакетов cookie заданной CGI-программой. По умолчанию после создания пакета cookie он будет возвращаться данному серверу по любому запрошенному броузером URL, включая URL, не содержащие CGI-программы. Рассмотрим, например, автомобильный Web-узел, схема которого показана на рис. 21.6.

```
|
|
|--Документы отдела продаж
|      └─CGI-программы отдела продаж
|        (каталог cgi-sales)
|
└--Документы производственного отдела
       └─CGI-программы производственного отдела
         (каталог cgi-engineering)
```

Рис. 21.6. Дерево каталогов Web-узла, рассчитанного на многоплановое использование

Имеет смысл размещать CGI-программы, связанные с отделом продаж, отдельно от CGI-программ производственного подразделения. Если бы сбытовая CGI-программа установила пакет cookie, то этот же пакет отправлялся бы и производственной CGI-программе, и наоборот. Такое положение вещей, конечно же, нежелательно, и создателям CGI-программ, предназначенных для обоих узлов, пришлось бы предпринять меры по координации действий во избежание конфликта по имени файлов cookie.

Чтобы обойти эту проблему, можно использовать аргумент -path функции cookie. С его помощью указывается путь (относящийся к URL верхнего уровня), по которому пакет cookie должен вернуться. Например, чтобы отправить пакет cookie, который будет возвращаться только к сбытовой CGI-программе, можно использовать следующий фрагмент кода:

```
# Cookie будет видимым только для сбытовой CGI-программы
$cookie=cookie{ -name => 'profile',
        -value => 'седан,люкс,двухдверный',
        -path => '/cgi-sales'};
print header(-cookie => $cookie);
```

По умолчанию файлы cookie возвращаются каждому узлу сервера, как если бы был использован параметр -path=>'/'. Для ограничения "области распространения" пакета cookie, т.е. чтобы он возвращался только заданной CGI-программе, в аргументе -path указывается URL CGI-программы.

```
# Обеспечиваем возврат пакета cookie только этой программе
$cookie=cookie( -name => 'profile',
        -value => 'седан,люкс,двухдверный',
        -path => script_name{)  J;
print header(-cookie => $cookie);
```

Вы должны помнить из материала предыдущего занятия, что функция script_name модуля CGI возвращает часть адреса URL текущей CGI-программы. С ее помощью можно быстро создать пакет cookie, который будет возвращаться только программе, установившей этот cookie в броузере.

Безопасность пакетов cookie

Для передачи некоторых пакетов cookie может потребоваться установить безопасное соединение с сервером. Используя аргумент -secure функции cookie, можно организовать оправку пакетов cookie с броузера только в том случае, если соединение является безопасным. С помощью следующего фрагмента программы организуется отправка броузеру номера счета пользователя. Пакет cookie, содержащий подобную информацию, должен передаваться *только* через безопасное соединение.

```
I Внимание!  Оправляйте только через безопасное соединение
$cookie=cookie( -name => 'account',
        -value => '00-12-3-122-1313',
        -secure => 1);
print header(-cookie => $cookie);
```

Чтение этого пакета cookie выполняется как обычно с помощью функции cookie. Если используется безопасное соединение и файл cookie установлен в броузере, то броузер при необходимости возвращает пакет cookie серверу.

```
# Считывание номера счета с броузера.
$account_number=cookie{'account');
```

Однако для установления факта безопасного соединения не следует слишком уж полагаться на этот метод. Не стоит также рассчитывать и на точность используемого номера счета. Помните, что именно пользователь управляет Web-броузером и его файлами cookie. Не исключено, что пакет cookie, содержащий закрытую информацию, может быть отправлен серверу через обычное (а не безопасное) соединение, да и номер счета может оказаться неверным.

Проблемы с файлами cookie

Прежде чем использовать файлы cookie в приложении, следует хорошо представлять себе проблемы, возможные при работе с ними. По этим причинам — а в будущем могут "всплыть" и другие неблагоприятные обстоятельства — желательно так проектировать свои Web-страницы и CGI-программы, чтобы они *не зависели* от файлов cookie.

Например, если вы используете cookie для хранения параметров настройки пользователя, то в случае, если cookie будут недоступны, программа должна взять стандартный набор параметров, Другими словами, предусмотрите меры "оборонительного характера".

Недолговечность файлов cookie

Как уже упоминалось, файлы cookie весьма недолговечны. Они могут быть случайно удалены из системы или изменены пользователем, или просто отвергнуты броузером без всяких на то причин.

Броузер может принять файл cookie, поработать с ним, а затем просто о нем "забыть", опять-таки без всяких видимых причин. Если вы установите перманентные файлы cookie с помощью аргумента -expire, броузер все равно может избавиться от них, не уведомив об этом пользователя.

Файлы cookie поддерживаются не всегда

Не секрет, что не все броузеры поддерживают файлы cookie. Стандарты Internet, которые применяются к HTTP- и Web-трафику, совсем не гарантируют, что броузеры должны поддерживать работу с cookie.

Сказанное выше вовсе не означает, что большинство броузеров не поддерживают cookie (все обстоит как раз наоборот). Такие броузеры, как Netscape (начиная с версии 1.1), Internet Explorer (все версии), Lynx, Opera и другие популярные Web-броузеры, поддерживают работу с файлами cookie. Однако в большинстве из них пользователь имеет возможность отключить поддержку cookie.

Даже если вы использовали функцию user_agent модуля CGI и выяснили, что ваш броузер должен поддерживать файлы cookie, не стоит полагаться на эту информацию.

Некоторые пользователи не любят cookie

Возможно, с заголовком этого раздела трудно согласиться: причем здесь "нелюбовь" к файлам cookie?

Путешествие по Web-пространству по своей сути является анонимным занятием. Как отмечалось на предыдущих занятиях, Web-сервер не запоминает состояния запроса, присланного броузером. Это означает, что сервер не может отследить адрес броузера и определить, когда тот подключался к нему в последний раз и какую информацию запрашивал.



> Помните: один броузер необязательно должен представлять одного пользователя. Броузером могут пользоваться много людей в доме, офисе, Internet-кафе или библиотеке. Установка (или изменение) файла cookie для одного пользователя может в действительности привести к его установке для нескольких пользователей сразу.

Файлы cookie могут быть использованы для отслеживания факта посещения конкретного Web-узла и сохранения параметров настройки, которые выбрали посетители. Поэтому если вас беспокоят вопросы конфиденциальности, эта информация может вас насторожить.

Например, гипотетический книжный магазин, о котором упоминалось выше (congo.com), может отслеживать, на каких названиях книг "щелкал" посетитель для получения более подробной информации, а затем использовать эти данные для составления соответствующих списков книг с последующим предоставлением их этому посетителю.

Внешне все выглядит прекрасно. Однако для тех, кого волнуют вопросы сохранения тайны, возникают две проблемы. Во-первых, кто угодно может отследить, какими книгами заинтересовался конкретный Web-посетитель. И если эта информация как-то связана с именем и адресом Web-посетителя (допустим, ее можно получить после анализа данных формы, заполненной на другом Web-узле, который совместно использует информацию с узлом congo.com), на этого "бедного" Web-посетителя может обрушиться поток электронной почты, имеющей отношение к теме интересующих его книг. Чем больше объем информации, совместно используемой узлами-сборщиками файлов cookie, тем более детализированными оказываются данные, которые могут быть собраны об этом Web-посетителе.

Помимо вопросов сохранения секретности возможны и другие негативные нюансы. Если первые две книги относились к категории "Компьютер", то Web-узел может уже не предлагать этому Web-посетителю книги из категорий "Романтика" и "Кулинария", т.е. Web-узел "записал" этого посетителя в одну категорию и отсек другие.



Возможно, вы будете удивлены тому, насколько часто файлы cookie используются для сбора и хранения информации и передаются вашему броузеру. Просто включите опцию подтверждения приема cookie в своем броузере и посетите любой популярный Web-узел.

Чтобы избежать вмешательства пакетов cookie в "частную" жизнь Web-посетителя, нужно предпринять определенные меры. Все Web-броузеры, поддерживающие cookie, предлагают средства для их отключения, а некоторые запрашивают разрешения у пользователя на установку cookie в броузер. Существуют специальные надстройки, предназначенные для фильтрации файлов cookie, отправляемых броузеру, и позволяющие их редактировать. Web-узлы спроектированы так, чтобы вы могли анонимно их посещать, и при этом никакая информация без вашего ведома не помешалась бы в файлы cookie.

Таким образом, некоторые пользователи считают, что файлы cookie посягают на конфиденциальность информации, поэтому, прежде чем использовать их, следует "еще раз хорошо подумать".

Резюме

На этом занятии вы узнали все о способах применения файлов cookie для хранения информации в броузере, которая впоследствии может быть передана на обработку некоторой CGI-программе. При этом возможна настройка, в результате которой пакеты cookie будут действовать на протяжении заданного периода времени или "видны" только для конкретного Web-сервера, или даже для заданных URL. Наконец, вы "окунулись" в проблемы, связанные с использованием файлов cookie, и познакомились со многими причинами, по которым не стоит с ними иметь дело.

Вопросы и ответы

Как поместить в файл cookie более одного значения?

Проще всего это сделать путем комбинации в одном файле cookie нескольких элементов, отделенных друг от друга символом-разделителем полей, как показано ниже на примере:

```
$cookie=cookie(-name  =>  'preferences',
 -value => 'bgcolor=blue,fgcolor=red,banners=no,java=no');
```

Позже, при считывании этого файла cookie, для разделения элементов можно использовать функцию split:

```
$cookie=cookie('preferences');
   @options=split(/,/, $cookie);
# Теперь создадим хэш из элементов.
# В качестве ключа используется часть строки до знака '='
foreach $option (@options) {
    ($key,$value)=split(/=/,$option)
    $Options{$key}=$value;
}
```

Как использовать файлы cookie для отслеживания того, на каких гиперссылках Web-страницы щелкнул пользователь?

Прежде чем ответить на этот вопрос, вы должны хорошо понимать, что некоторые пользователи считают применение файлов cookie посягательством на конфиденциальность их информации. А теперь рассмотрим общий метод.

1. Создайте гиперссылки <A HREF> таким образом, чтобы они обеспечивали запуск CGI-программы, в качестве параметра target которой передается *действительный* URL:

   ```
   <A HREF="http://server/cgi/redirect.pl?target=http://www.congo.com">
   Congo</A>
   ```

2. В этом примере программа redirect.pl для получения реального URL (http://www.congo.com) из параметра target должна использовать функцию param модуля CGI:

   ```
   $target_url=param('target');
   ```

3. Затем создается файл cookie, значение которого равно URL. Имя cookie может быть любое, по которому cookie позже можно будет найти (здесь использовано имя tracker):

   ```
   $tracking_cookie=cookie(-name     =>  'tracker',
                           -value    => $target_url,
                           -expires  => '+1w');
   ```

4. После этого функция redirect отправляет URL броузеру вместе с cookie:

   ```
   print redirect(-uri    => $target_url,
                  -cookie => $tracking_cookie);
   ```

   Позже, когда броузер вернется к вашему Web-узлу, CGI-программе будет передан файл cookie с именем tracker, содержащий URL, который посетил пользователь во время пребывания на вашем узле.

Можно ли передать cookie в процессе перенаправления броузера на другую Web-страницу?

Конечно. Функция redirect модуля CGI (подобно функции header) может также иметь аргумент -cookie:

```
my $cookie=cookie(-name  => 'target',
                  -value => 'переадресация на foo.html');
print redirect(-uri => "http://www.server.com/foo.html",
            -cookie => $cookie);
```

Семинар

Контрольные вопросы

1. Почему использование перманентных файлов cookie иногда не оправдывает ожиданий?

   а) броузеры могут отвергнуть файлы cookie;

   б) файлы cookie могут потеряться во время обновления программного обеспечения;

   в) пользователи могут отключить поддержку cookie в своих броузерах.

2. Какое значение должен иметь аргумент -expire функции cookie, чтобы файлы cookie были активны в течение одной недели, начиная с текущего момента времени?

   а) +7d;

   б) +lw;

   в) +10080m.

3. Почему некоторые люди видят в использовании файлов cookie проблему посягательства на конфиденциальность их информации?

   а) файлы cookie могут быть использованы для отслеживания ссылок на Web-страницы, которые посещал пользователь;

   б) собранная с помощью файлов cookie информация может быть использована для создания досье на пользователей;

   в) файлы cookie можно использовать для селектирования информации, передаваемой пользователю.

Ответы

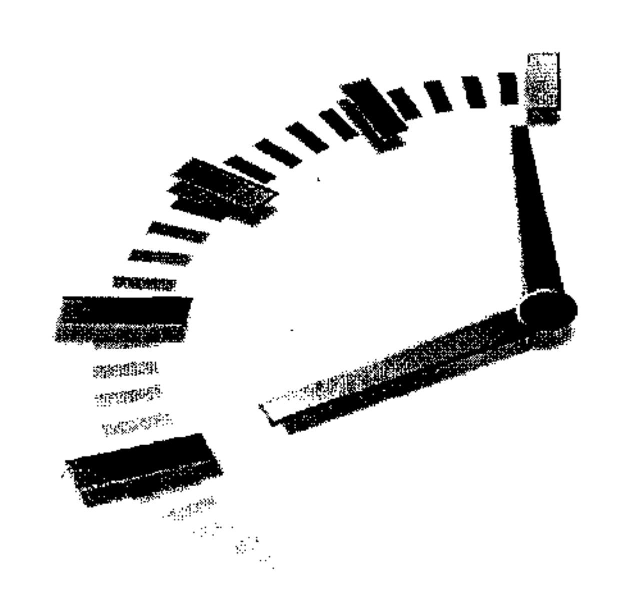
1. Все пункты содержат правильные ответы.
2. Правильным будет либо вариант а), либо в). Аргумент +lw неверен.
3. Все пункты содержат правильные ответы.

Упражнения

Расширьте функциональные возможности CGI-программы, устанавливающей цвет фона. Добавьте в программу возможность выбора шрифта, а также изображения для помещения его на Web-страницу путем изменения содержимого дескриптора <IMG>.

22-й час

Отправка электронной почты из CGI-программ



Несомненно, бороздя по просторам Web, вам приходилось заполнять форму, данные которой использовались для отправки электронной почты. Такие формы обычно применяются для составления списков рассылки, сообщений о неисправностях, обслуживания клиентов и пр.

На этом занятии вы узнаете, как из Perl-программ отправлять электронную почту, и рассмотрите пример небольшой Web-страницы, которую можно использовать для генерирования сообщений электронной почты. Как применять эту страницу — это уж решать вам.

Основные темы этого занятия.

- Как работает электронная почта Internet.
- Как послать электронное сообщение в системе UNIX и в других отличных от нее операционных системах.
- Как настроить Web-форму для отправки электронной почты.

Основы работы электронной почты Internet

Прежде чем дать волю своей фантазии на ниве программирования по части отправки сообщений электронной почты с помощью средств Perl, необходимо разобраться в том, как организована работа электронной почты в Internet.

Еще до рождения Perl, когда службы Web не было и в помине (т.е. в "мыслях" у NCSA) и модемы работали очень медленно, электронная почта была реализована через систему, именуемую *UNIX-to-UNIX copy (UUCP)*. Когда вы посылали электронное сообщение в той старой системе, локальный компьютер "упаковывал" его и передавал дальше, к следующему компьютеру в цепочке, которая снова "упаковывала" его и передавала очередному звену. Каждая система в цепи добавляла к "транзитному" сообщению немного "своих" данных, которые означали, что сообщение обработано и передано дальше (рис. 22.1).

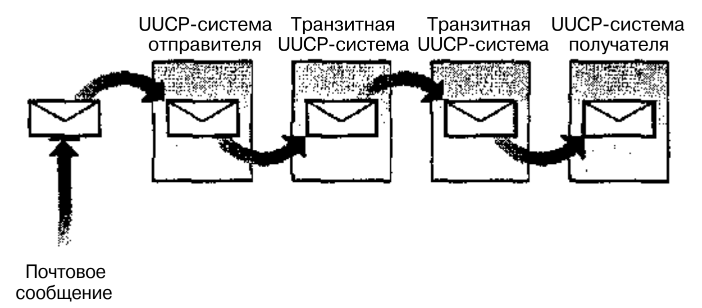


Рис. 22.1. Передача почты от системы к системе

Этот метод обмена почтовыми сообщениями по вполне очевидным причинам называется ***передачей*** *сообщения с промежуточным хранением* (store and forward). Система UUCP была с тех пор заменена более современными системами, но основной метод передачи с промежуточным хранением по-прежнему остался в силе. При отправке почты со своего компьютера ее "подхватывает" другая система и передает еще одной системе, которая затем передает почту дальше, и так продолжается до тех пор, пока ее не получит система, для которой предназначено сообщение.

Современные протоколы в корне изменили процесс передачи сообщений. Самым популярным является простой протокол пересылки почты (Simple Mail Transport Protocol — SMTP), который используется для отправки почты по цепи (рис. 22.2). Для получения почты в пункте назначения обычно используется либо почтовый протокол (Post Office Protocol — POP), либо протокол доступа к сообщениям Internet (Internet Message Access Protocol — IMAP). Пока остановимся на работе протокола SMTP.

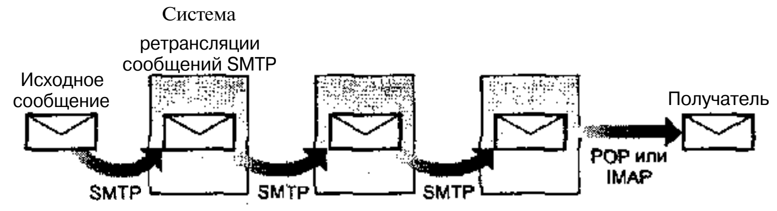


Рис. 22.2. Для пересылки и получения почты используются различные протоколы

Отправка почтового сообщения

Чтобы послать сообщение, вам потребуется *агент передачи почты (Mail Transport Agent*, или *MTA)* и компьютер, обеспечивающий функционирование протокола SMTP. Объясним поподробнее.

Агент передачи почты — это программа, которая располагается на компьютере пользователя. Обычно она поставляется вместе с операционной системой и отвечает за принятие сообщений электронной почты и корректную их ретрансляцию. MTA обычно настраивается при установке операционной системы. Одна из распространенных MTA-программ в системах UNIX называется `sendmail`. Программа `sendmail` принимает почтовое сообщение и решает, как доставить его по назначению.

Чтобы послать почтовое сообщение в среде UNIX, используйте следующий синтаксис командной строки:

```
$ /bin/echo "Subject:Test\n\nПривет, мир!" | sendmail foo@bar.com
```

В этом фрагменте организуется отправка короткого сообщения по адресу foo@bar.com. Программа sendmail берет на себя все трудные этапы работы: принимает решение о выборе метода передачи сообщения, обрабатывает возвращаемую почту и т.д.

В операционных системах Microsoft Windows или Macintosh нет встроенной программы МТА. К счастью, модуль Perl позволяет отправлять почту напрямую. Модуль Net::SMTP может посылать почту без посредничества программы МТА, но в этом случае вам нужно знать имя компьютера, обеспечивающего ретрансляцию почты с помощью протокола SMTP. Обычно это имя указывается провайдером в карточке регистрации при получении доступа к Internet в графе "Mail Host", или сервер SMTP. Узнайте имя своего почтового сервера и запишите его где-нибудь: оно потребуется вам чуть позже.



Провайдер может сообщить вам несколько имен серверов, управляющих процессами отправки и получения почтовых сообщений. Для тем, обсуждаемых на этом занятии, вам понадобится узнать имя SMTP-сервера, отправляющего почту.

Помните: программе, которая отправляют почту по протоколу SMTP, понадобится корректно работающий SMTP-сервер — в противном случае процесс отправки почты работать не будет.



Имя SMTP-сервера зависит от того, откуда вы отправляете почту. Если вы делаете это из дома, имя сервера обязательно сообщит вам провайдер услуг (Internet service provider — ISP). Если вы отправляете почту, используя учетную запись на арендуемом Web-сервере, имя SMTP-сервера можно узнать на одной из Web-страниц этого сервера. Обычно почтовые серверы не принимают сообщения, посланные с незнакомых для них систем.

Немного о правилах хорошего тона

В следующем разделе вы узнаете о новой функции send_mail, которую можно использовать для отправки почты с помощью Perl-программы. Это средство весьма полезно и в то же время очень опасно. Отправка кому-либо почты — это в некотором роде посягательство на его частную жизнь и собственность. Представьте такую картину: вы просите получателя почты выделить вам некоторое время и дисковое пространство. Вы также просите все промежуточные системы, расположенные между вашей и системой получателя, передать эту почту для вас. Не слишком ли много просьб для постороннего лица?

Ниже перечислены некоторые "правила хорошего тона", или сетевого этикета, которые следует соблюдать при отправке почты с помощью Perl-программ или любого другого средства.

- Сначала проверьте свою программу с использованием хорошо известных адресов (например, своих собственных) и маленьких сообщений. Неприятности и так подстерегают на каждом шагу — постарайтесь не создавать новых.

- Не отправляйте по электронной почте сообщения коммерческого характера *без специального запроса.* Непрошеная почта — обычно называемая ***спэмом*** — стала настоящей проблемой для пользователей Internet. Лишь немногие не возражают против получения такого вида почтовых сообщений. Остальных же это по меньшей мере раздражает, а то и просто возмущает. Корпорации, злоупотребляющие возможностями электронной почты, непременно навлекают на себя гнев множества пользователей Internet. Интересуясь адресом электронной почты, всегда спрашивайте, согласен ли этот человек впоследствии получить от вас сообщение. Уважайте также желание многих пользователей не попадать в посторонние списки почтовой рассылки.

- Не отправляйте по электронной почте за один раз слишком объемные сообщения (даже если они были затребованы). Во-первых, буфер вашего локального почтового сервера может переполниться. При этом местный провайдер наверняка временно закроет вашу учетную запись, чтобы справиться с неполадками. Во-вторых, если ваше сообщение вызовет перегрузку почтового сервера получателя, удаленный провайдер может просто заблокировать всю электронную почту, поступающую из вашего домена. Потеря возможности посылать что-либо таким крупным доменам, как aol.com, hotmail.com, может плачевно сказаться на вашем бизнесе. Более того, вполне вероятно, что ваш провайдер закроет вашу учетную запись и выставит вам дополнительный счет.

- Позаботьтесь об указании корректного обратного адреса в заголовках From: и Reply To: сообщения электронной почты. Существует возможность использования Perl для подделки электронной почты, но подделка не содержит цепочки адресов почтовых серверов, через которые прошло сообщение. Однако учтите, что вы можете нажить себе серьезные неприятности, подделывая почтовые сообщения.

- Всегда используйте свой собственный почтовый сервер. Злоупотребление почтовыми серверами других провайдеров быстро приведет к закрытию вашей учетной записи и, возможно, даже к предъявлению вам дополнительного счета.

- Не отправляйте ничего не подозревающим людям огромных или большого числа маленьких сообщений. Это называется *почтовой бомбардировкой* (mailbombing) и, скорее всего, закончится аннулированием вашей учетной записи провайдером и может вызвать неприятности в области правовых норм.

Практически все эти советы являются не просто "правилами хорошего тона", или сетевого этикета. Нарушение некоторых из перечисленных правил может привести к прекращению обслуживания вас провайдером Internet и/или повлечь правовую ответственность за ущерб, нанесенный получателям вашей почты. При создании учетной записи провайдер, скорее всего, предупредит вас о том, что любой из вышеперечисленных факторов послужит основанием для ее аннулирования, и, возможно, вы будете нести личную ответственность за нанесенный ущерб.

Золотое правило: быть консерватором, когда вы отправляете что-то другим, и либералом по отношению к тому, что вы получаете.



Для Internet характерна очень долговременная коллективная память. Тех, кто действительно злоупотреблял почтовой службой, помнят очень долго и не хотят иметь с ними дела. Помните, что испорченную репутацию очень трудно потом исправить.

Программные средства организации почтовой службы

Цель следующих разделов — написать на Perl короткую функцию, которую можно использовать в *CGI-программах* для отправки почтовых сообщений. Однако существует одна проблема: работа такой функции зависит от того, имеется ли в вашей системе агент передачи почты (такой, как `sendmail`), либо программа напрямую должна

отправлять почту по протоколу SMTP ближайшему серверу. Поэтому бегло просмотрите следующие разделы и решите, какой из них вам понадобится для реализации конкретной программы.

Отправка почты в системах UNIX

Если вы работаете в системе UNIX и у вас корректно настроена программа sendmail (а так оно и должно быть!), это именно тот раздел, который вам нужен. Если же какая-либо из необходимых составляющих (система UNIX или программа sendmail) отсутствует и вы читаете этот раздел лишь из любознательности, то заслуживаете всяческих похвал. Однако в этом случае функция, представленная в листинге 22.1, скорее всего, вам не подойдет.



Даже если вы работаете в системе UNIX, вам все равно стоит прочесть следующий раздел, "Отправка почты не из системы UNIX". В нем рассматривается новый метод использования объектно-ориентированного модуля.

Листинг 22.1. Функция send_mail

```
1:  # Функция для отправки почты с помощью программы MTA (sendmail)
2:  sub send_mail {
3:     my($to, $from, $subject, @body)=@_;
4:
5:     # Измените следующую строку в соответствии с вашей системой
6:     my $sendmail="/usr/lib/sendmail -t -oi -odq";
7:
8:     open(MAIL, "|$sendmail") ||
          die "Ошибка при запуске программы sendmail: $!";
9:      print MAIL<<END_OF_HEADER;
10: From: $from
11: To: $to
12: Subject: $subject
13:
14: END_OF_HEADER
15:    foreach (@body) {
16:            print MAIL "$_\n";
17:     }
18:     close(MAIL);
19: }
```

 Проведем анализ программы.

- *Строка 6.* Переменной `$sendmail` присваивается значение полного пути к программе sendmail и необходимые аргументы для ее запуска. Учтите, что программа sendmail может находиться в другом каталоге вашей системы или иметь другие аргументы.
- *Строка 8.* Открывается конвейер для записи информации в стандартный входной поток программы sendmail, полный путь и параметры которой указаны в переменной Ssendmail. Конвейеру назначается дескриптор файла MAIL.

- *Строки 9~14.* Заголовок почтового сообщения выводится в дескриптор MAIL
- *Строки 15—17.* Тело сообщения записывается в дескриптор MAIL. В конец каждой строки добавляется символ \n.

Чтобы протестировать эту функцию, просто вызовите ее с набором из четырех аргументов:

```
@body=("Пожалуйста, прекратите интриги!", "Спасибо!*);
send_mail('president@whitehouse.gov', 'owner@geeksalad.org',
    'Taxes', @body);
```

При вызове этой функции считается, что программа sendmail должна уже быть корректно установлена и настроена для работы в вашей системе. Если она отсутствует, переходите к следующему разделу, "Отправка почты не из системы UNIX", поскольку представленное там решение должно работать также и под управлением системы UNIX.

Вам нужно будет изменить значение переменной $sendmail в соответствии с действительным расположением программы sendmail в вашей системе. Она может находиться в одной из таких папок, как /usr/lib, /usr/sbin и /lib, или в любой другой. Для поиска используйте команду whereis sendmail.



Если что-то работает не так, как вы ожидали, убедитесь в корректной работе службы электронной почты вашей системы. Попытайтесь отправить контрольное сообщение с помощью одной из почтовых утилит, как, например, mail или pine. Если эти утилиты не работают надлежащим образом, то маловероятно, что программа sendmail установлена корректно. Приступая к работе, вам придется устранить эту проблему или использовать метод, описанный в следующем разделе.

В листинге 22.1 программа sendmail запускается с использованием описанных ниже параметров, причем вы можете изменить их по своему усмотрению.

-t	Поля заголовка (From, To, Subject и т.д.) берутся из входных данных, а не из командной строки.
-oi	Единственный символ точки, расположенный в начале строки, не считается признаком конца сообщения. Этот ключ позволяет предотвратить случайное завершение сообщения.
-odq	Программа sendmail помещает сообщения в очередь, а не пытается отправить их немедленно. При желании эту опцию можно не использовать. Но если будет отправлено одновременно слишком много сообщений, ваша почтовая система может быть перефужена запросами. Более корректно все-таки использовать ключ -odq.

Остальная часть функции send_mail() говорит "сама за себя".

Отправка почты не из системы UNIX

При работе под управлением Windows и других операционных систем, в которых нет встроенной программы-агента МТА типа sendmail, вам не избежать определенных трудностей. Программы МТА не так уж просты, и попытка скопировать их действия с помощью нескольких строк Perl оказывается не из легких. Тем не менее это осуществимо.

Сначала хорошая новость: в Perl есть специальный модуль Net::SMTP, который позволяет отправлять почту из операционныгх систем любого типа, в которых могут работать Perl-программы. Используя этот модуль, можно отправить почту без особых усилий.

Теперь о плохом: этот модуль не входит в стандартную поставку Perl. Вы должны загрузить его самостоятельно и установить на том компьютере, откуда собираетесь отправлять почту. Модуль Net::SMTP является частью пакета libnet, который содержит все виды полезных модулей, обеспечивающих работу в сети. Пакет libnet находится на прилагаемом к этой книге компакт-диске.



В приложении "Инсталляция модулей в Perl" подробно описан процесс установки модулей в Perl под управлением таких операционных систем, как UNIX, Windows и Macintosh. Кроме того, вы получите инструкции по установке собственных копий модулей на случай, если системный администратор не захочет устанавливать общую копию.

Функция send_mail для систем без программы MTA представлена в листинге 22.2. Она включает несколько необычный (новый) синтаксис, с которым вы еще незнакомы. Обязательно прочитайте последующие пояснения.

Листинг 22.2. Функция send_mail для систем без MTA

```
1: # Функция для отправки почты для систем без программы MTA
2: sub send_mail {
3:    my($to, $from, $subject, @body)=@_;
4:
5:     use Net::SMTP;
6:
7:    # Вам придется изменить следующую строку,
8:    # чтобы указать правильное имя почтового сервера
9:     my $relay="relayhost.yourisp.com";
10:    my $smtp = Net::SMTP->new($relay);
11:   die "Невозможно открыть соединение: $!" if (! defined $smtp);
12:
13:     $smtp->mail($from);
14:    $smtp->to($to);
15:
16:     $smtp->data();
17:     $smtp->datasend("To: $to\n");
18:     $smtp->datasend("From: $from\n");
19:    $smtp->datasend("Subject: $subject\n");
20:     $smtp->datasend("\n");
21:    foreach(@body) {
22:         $smtp->datasend("$_\n");
23:    }
24:   $smtp->dataend(); # Обратите внимание на орфографию: нет "s"
25:    $smtp->quit;
26: }
```

Проведем анализ программы.

- *Строка 5.* Подключается модуль Net::SMTP, который позволяет намного упростить отправку почты.
- *Строка 10.* Создается объект Net::SMTP, связанный с соответствующим почтовым сервером, адрес которого был указан в строке 9.

- *Строки 13—23.* Серверу электронной почты отправляются заголовки и тело сообщения. Для получения более подробной информации см. пояснения по функциям Net::SMTP, приведенные ниже.

Чтобы протестировать эту функцию, достаточно вызвать ее с четырьмя аргументами, представляющими отдельные части почтового сообщения:

```
@body=("Пожалуйста, прекратите интриги!", "Спасибо!");
send_mail('president@whitehouse.gov', 'owner@geeksalad.org',
    'Taxes', @body);
```

Первое, что может удивить вас в этой функции, это строка `$smtp = Net::SMTP->new($relay)`. При ее выполнении создается нечто, именуемое *объектом.* Объект — это не скаляр, не случайные данные ("мусор") и не массив, это нечто совсем иное. Значение в переменной $smtp представляет соединение с программой пересылки почты, которой можно управлять. Представьте себе объект как значение особого вида, которое позволяет вызывать функции, имеющие к нему отношение.

Следующей "странностью" для вас может оказаться строка `$smtp->mail($from)`. Элемент -> соединяет объект, заданный слева, с функцией, указанной справа от него. Поэтому mail — это вызов функции, в котором используется объект `$smtp`, созданный на предыдущей строке.

Для использования модуля `Net::SMTP` вам необязательно вникать во все подробности синтаксиса объекта — достаточно его соблюдать. Ниже перечислены функции, которые можно использовать вместе с объектом `Net::SMTP`.

- `$smtp->mail(addr)`. Функция mail указывает, кто является отправителем почты. Здесь вполне можно сообщить "не свой" адрес.

- `$smtp->to(addr)`. Функция to определяет, кому вы отправляете сообщение. При вызове функции to со списком имен каждый адресат получит копию данного почтового сообщения. Имена получателей необязательно должны содержаться в теле сообщения, например, если вы их поместите в поле `BCC:`.

- `$smtp->data()`. Функция data означает, что вы готовы отправить само сообщение,

- $smtp->datasend(data). Эта функция отправляет действительный текст сообщения. Вы должны явно вывести обязательные поля заголовка (`To:`, From: и т.д.), а такие поля, как Date: и Received:, генерируются автоматически. Между заголовком и телом сообщения необходимо вывести пустую строку: `$smtp->datasend("\n")`. За пустой строкой следует тело сообщения, которое также посылается с помощью конструкции $smtp->datasend().

- $smtp->dataend(). Функция dataend означает, что вы завершили отправку тела сообщения. Сообщение не будет послано до тех пор, пока не обозначится конец сообщения.

- `$smtp->quit()`. Эта функция завершает сеанс связи с SMTP-сервером.

Отправка почты из Web-страницы

Теперь в вашем распоряжении есть способ отправки электронной почты программными средствами с помощью функции send_mail(}, а уж организовать это "мероприятие" из Web-страницы — пара пустяков! Вам просто нужно спроектировать

страницу и написать CGI-программу, координирующую "совместные" действия. Пример HTML-формы, предназначенной для отправки электронной почты, представлен в листинге 22.3. Эта форма не отличается особой изысканностью, но при желании вы можете "украсить" ее по своему вкусу.

Листинг 22.3. HTML-форма для отправки электронной почты

```
<!--Предполагается наличие программы /cgi-bin/mailer.cgi-->
<FORM METHOD=POST ACTION="/cgi-bin/mailer.cgi">
Ваш адрес: <INPUT TYPE=text NAME=return_addr><BR>
Тема: <INPUT TYPE=text NAME=subject><BR>
<BR>
Сообщение:<BR>
<TEXTAREA NAME=body ROWS=20 COLS=60 WRAP=hard>
Введите текст сообщения
</TEXTAREA>
<BR>
<INPUT TYPE=SUBMIT VALUE="Отправка сообщения">
</FORM>
```

CGI-программа для отправки почты ненамного больше. Вот как она выглядит:

```
#!/usr/bin/perl -v
use strict;
use CGI qv(:all);
use CGI::Carp qw(fatalsToBrowser);

#
# Вставьте сюда функцию send_mail из
# листинга 22.1 или 22.2!
#

print header;
my $return=param("return_addr");
if (! defined $return or ! $return) {
        print "Необходимо указать адрес e-mail<P>";
        exit;
}
my $subject=param("subject");
if (! define $subject or ! $subject) {
        print "Необходимо указать тему сообщения<P>";
        exit;
}
# Замените этот адрес адресом получателя
# сообщений электронной почты
send_mail('webmaster@myhost.com',
          param($return),
          param($subject),
          param("body"));

print "Почта отправлена.";
```

Следует обратить внимание на несколько моментов в этой маленькой программе. Во-первых, чтобы обеспечить работоспособность этой программы, вы должны вставить функцию `send_mail`, текст которой приведен либо в листинге 22.1, либо в листинге 22.2.

Используйте вариант, который работает лучше и больше вам подходит. Во-вторых, обратите внимание на то, что адрес, указываемый в поле To:, жестко "зашит" в программе — в виде адреса webmaster@myhost.com. Вам придется изменить этот адрес, поскольку по нему будут отправляться сообщения. Причина, по которой адрес получателя не берется из поля формы, совсем проста: не стоит разрешать неизвестным пользователям отправлять сообщения по электронной почте с помощью Web-формы. Если кто-нибудь будет злоупотреблять использованием вашей формы и посылать "направо и налево", скажем, неприличные сообщения, то вас могут обвинить как инициатора сообщений такого рода. Вряд ли вам захотелось бы очутиться в такой ситуации.

Если нужно, то, используя только одну Web-форму, можно организовать отправку сообщений по нескольким адресам. Для этого создайте раскрывающийся список получателей или используйте кнопки переключателя:

```
<INPUT TYPE=radio NAME=target Value=1 CHECKED>Отдел технической поддержки
<INPUT TYPE=radio NAME=target Value=2>Отдел продаж
<INPUT TYPE=radio NAME=target Value=3>Юридический отдел
```

Затем в CGI-программе используйте следующий программный код:

```
$formtarget=param('target');
%targets=( 1=> 'support@myhost.com',
           2=> 'sales@myhost.com',
           3=> 'legal@myhost.com');
if (exists($targets{$formtarget})) {
        $target=$targets{$formtarget};
} else {
        $target='webmaster@myhost.com';
}
print $target;
```

Какой бы способ вы ни выбрали, не разрешайте передавать почтовые сообщения из Web-формы по произвольно задаваемому адресу To:. Вместо этого передайте CGI-программе какое-нибудь безопасное значение (в данном примере мы использовали число в диапазоне 1-3), а затем интерпретируйте его соответствующим образом. Обратите внимание, как в нашем примере с помощью директивы else обрабатывается некорректное значение (на всякий случай).

Контроль адресов электронной почты

Возможно, вы заметили, что CGI-программа не пытается установить, достоверен ли адрес, введенный пользователем. На это есть причина: сие невозможно!

Такой поворот дел может удивить вас.

Одна из "святых заповедей" разработки систем электронной почты в Internet — знать, достоверен ли адрес получателя. На это есть короткая отповедь: это невозможно!

О трудностях в решении этого вопроса можно судить, обратившись к рис. 22.1 и 22.2, приведенным в начале данного занятия. С позиции отправителя совсем не виден конец всей цепи доставки сообщения. Она должна полностью передать сообщение второй системе в цепи, которая затем передаст его третьей, и т.д. Задержка между передачами "эстафетной палочки" весьма значительна. И важно то, что инициирующая система не имеет никакого контроля над сообщением после того, как она "сбыла его с рук".

Стандартный подход — постараться избавиться от явно неверных адресов, но, к сожалению, нет способа для утверждения, что данный адрес недействителен. Стандарт Internet для адресов электронной почты — RFC 822 — содержит шаблон для стандартных адресов электронной почты. Однако нельзя не отметить, что некоторые совер-

шенно корректные с точки зрения стандарта RFC-822 адреса оказываются недействительными, а другие, нарушающие этот стандарт, — вполне действительны.

Попытки написать регулярные выражения, чтобы с ними можно было сравнивать адреса электронной почты, ни к чему не привели. Например, выражение /^[\w.-]+\@([\w.-]\.)+\w+$/ выглядит вполне подходяще. Ему даже можно поставить в соответствие такие адреса, как me@somewhere.com. Однако это выражение "забракует" следующие вполне рабочие адреса:

```
*§qz.az
clintp!sol2!westwood@dec.net
relay%me@host.com
"barney&fred"@flintstones.net
```

Одно регулярное выражение, которое соответствует адресам электронной почты, удовлетворяющим стандарту RFC-822, содержит 4 700 символов — слишком много, чтобы приводить его в этой книге и ожидать от вас готовности его использовать. Причем следует учесть, что оно совпадает далеко не со всеми действующими адресами в Internet.

Так что же делать?

Единственный способ определить достоверность некоторого адреса электронной почты — послать по этому адресу сообщение и ожидать ответа. Если по каким-то причинам вам понадобится иметь гарантии существования получателя с таким-то адресом, например для отправки ему сообщений в будущем, пошлите ему сообщение с просьбой ответить отправителю. Получив ответ, вы будете знать, что имеете дело с достоверным адресом электронной почты.

Резюме

На этом занятии вы узнали, как отправлять сообщения электронной почты из Web-страницы. Были рассмотрены две версии функции send_mail(), которые можно использовать в Perl-программе для отправки почты. При этом вы постигли основы работы электронной почты в Internet и познакомились с базовыми правилами сетевого этикета.

Вопросы и ответы

Можно ли использовать информацию от броузера для определения адреса электронной почты посетителя?

Хотя такая возможность кажется замечательной — с ее помощью мы бы избавились от ошибок ввода адресов электронной почты, — она попросту невозможна. У броузера нет адреса электронной почты пользователя. Значение, возвращаемое функцией remote_host модуля CGI, может в действительности не быть адресом, по которому пользователь получает электронную почту. Функция remote_user (если вы используете безопасные Web-транзакции), скорее всего, не будет содержать полного электронного адреса пользователя. И помните, броузер вполне может быть "недобросовестным" по части информации такого рода, к тому же некоторые дополнительные модули для броузеров Netscape и Internet Explorer поступают именно таким образом.

Кроме того, учтите, что пользователь может работать с Web-броузером в библиотеке, дома у приятеля, на работе или в Internet-кафе, поэтому адрес броузера даже удаленно никак не связан с адресом электронной почты пользователя.

Можно ли проверить адрес электронной почты?

Можете попытаться. Например, в большинстве современных адресов содержится символ @ (коммерческого *at),* и его можно использовать для проверки. Однако адреса на локальном компьютере {например, postmaster, root) не содержат символа @.

Я опробовал CGI-программу отправки электронной почты и получил в сообщении строку From `nobody...`?

Ах, да. Программа sendraail регистрирует идентификатор отправителя сообщения. В действительности "отправителем" является сам Web-сервер. Web-сервер часто работает со специальным идентификатором пользователя (ID) — nobody, Web, `httpd` или (не дай Бог!) root - и именно этот адрес указывается в заголовке почтового сообщения. Ничего страшного. Если в качестве части заголовка вы обеспечите соответствующую строку From:, то именно эта строка и будет видна на экране у получателя.

Как к сообщению электронной почты присоединить файл?

Вам стоит заглянуть в MIME-модули, находящиеся в CPAN.

Семинар

Контрольные вопросы

1. Какое действие выполняет строка `$foo=Net::SMTP->new('mailhost')`?

   (Если вы до сих пор не прочитали раздел, "Отправка почты не из системы UNIX", то, возможно, вам стоит сделать это сейчас.)

   а) вызывает синтаксическую ошибку;

   б) создает объект, именуемый $foo, который представляет подключение к почтовому SMTP-серверу;

   в) включает в текущую программу модуль Net: :SHTP.

2. Какой из приведенных ниже вариантов не является (потенциально) действительным адресом электронной почты?

   а) `foo!bar!baz!quux`.

   б) `" "@bar.com`.

   в) `stuff%junk!"Wowzers"!foo.com!blat`.

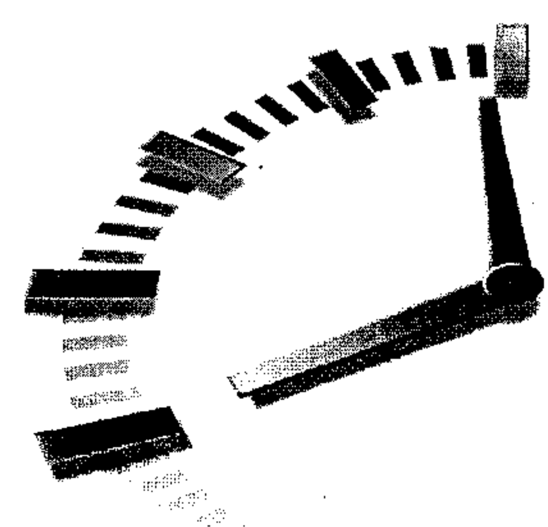
Ответы

1. Правильный ответ — вариант б). Если в качестве ответа вы выбрали пункт а), то вы либо допустили опечатку, либо работаете с версией Perl 4. Вариант в) неверен, поскольку представляет собой описание действия оператора use `Net::SMTP`.

2. Сложный вопрос. Все они являются потенциально допустимыми адресами электронной почты.

Упражнения

- Внесите следующие изменения в простую CGI-программу отправки электронной почты.
  - Выберите данные о броузере пользователя и вложите их в тело сообщения.
  - Отправьте пользователю копию сообщения с правилами сетевого этикета (не забудьте сообщить ему, что вы именно так и поступаете при работе с реальными Web-узлами). Следует также иметь в виду, что эти сведения всегда могут вызвать недовольство получателя.
- Разрешите пользователю "просмотреть" построенное им сообщение перед отправкой. Для этого вам придется использовать один из методов, описанных на 19-м занятии, "Сложные формы", чтобы сделать данные из первой страницы (экран ввода текста сообщения электронной почты) доступными для второй страницы (экран проверки правильности почтового сообщения) и в конечном итоге для CGI-программы отправки почты.



23-й час

Push-технология и счетчики посещений Web-страниц

На этом занятии вы познакомитесь с некоторыми распространенными методами CGI-программирования. Они помогут сделать ваши Web-страницы более интересными за счет использования анимации и тем самым оставить всех конкурентов далеко позади.

Основные темы этого занятия.

- Использование push-технологии сервером для обновления Web-страниц.
- Создание счетчиков посещений Web-страниц.
- О proxy-сервере и кэш-памяти.

Что такое push-технология

Традиционная Web-технология является по своей природе постраничной. Она мало приспособлена для просмотра больших документов, особенно когда речь заходит о медленных каналах связи.

В качестве примера рассмотрим Web-страницу, запускающую CGI-программу, которой требуется для выполнения очень много времени. Скажем прямо, шанс, что пользователь дождется окончания загрузки такой страницы, очень невелик.

Во-первых, броузер может прервать сеанс связи из-за окончания интервала ожидания завершения работы CGI-программы. Броузеры обычно ожидают результатов примерно в течение 90 секунд, а затем отображают сообщение, уведомляющее о том, что данный узел недоступен.

Во-вторых, CGI-программа время от времени может выводить сообщение: "Выполнено 10% работы", а несколькими мгновениями спустя: "Выполнено 20% работы" и т.д. Вывод подобных сообщений — дело хорошее, но дело в том, что они могут не появляться через одинаковые интервалы времени (из-за буферизации), и в результате вы получите чрезвычайно медлительную Web-страницу.

Вероятно, было бы лучше, чтобы сообщения, уведомляющие о состоянии загрузки Web-страницы, выводил броузер (рис. 23.1).

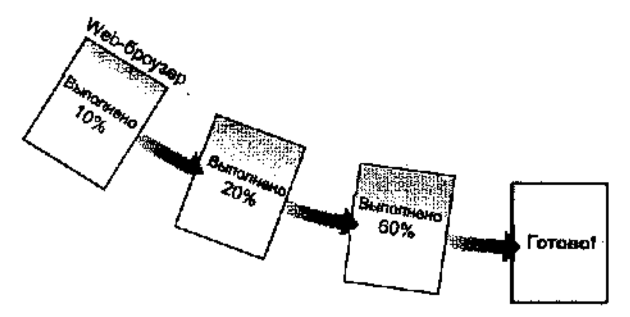


Рис. 23.1. Броузер, отображающий текущее состояние процесса загрузки Web-страницы

В *push-технологии {server push),* или технологии выталкивания страниц сервером, используется преимущество того факта, что броузеры могут получать страницы по разделам и повторно отображать эти страницы в нужной последовательности, как если бы вы сами последовательно загрузили различные страницы.



На момент написания этой книги броузер Internet Explorer компании Microsoft еще не поддерживал протоколы, необходимые для реализации push-технологии серверами. И это весьма печально, поскольку использование этой возможности предоставляет простой способ выполнения анимации содержимого Web-страниц. Для создания Web-страниц, корректно отображающихся в Internet Explorer или других броузерах, не поддерживающих push-технологию, следует использовать другие методы, например метод вытаскивания страниц самим клиентом (client pull).

Организация работы сервера в режиме выталкивания страниц

Чтобы обеспечить работу Web-сервера в режиме выталкивания страниц, его нужно соответствующим образом настроить. Для этого необходимо запустить CGI-программу на сервере в режиме, в котором не анализируются выводимые ею заголовки. В этом случае сервер не требует от CGI-программы вывода заголовков и пересылает данные "как есть" напрямую броузеру. Обычно Web-серверы анализируют результаты работы CGI-программ, чтобы убедиться в их корректности (вот откуда возникает ошибка с кодом 500!). При отсутствии анализа заголовков CGI-программа посылает результаты своей работы прямо броузеру, как показано на рис. 23.2.

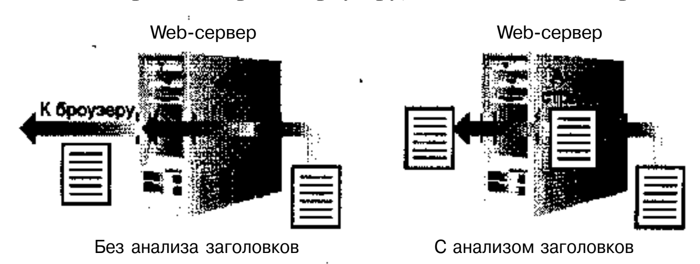


Рис. 23.2. Не анализируемые данные проходят через сервер без проверки

Как запустить свою CGI-программу и "убедить" сервер оставить заголовки нетронутыми — зависит от самого Web-сервера. Работая, например, с Web-сервером Apache, *обычно* достаточно, чтобы имя файла CGI-программы начиналось с префикса nph-. Например, файл с именем `push.cgi` будет обрабатываться как CGI-программа с анализируемыми заголовками, а файл с именем `nph-push.cgi` — как CGI-программа без анализа заголовков. Однако администратор Web-сервера может изменить описанную выше схему наименования CGI-программ.

Под управлением сервера Internet Information Server (IIS) компании Microsoft все CGI-программы выполняются как программы без анализа заголовков; функция header модуля CGI обычно скрывает этот факт от вас. Поэтому для работы Web-сервера IIS в режиме выталкивания страниц не нужно вносить никаких изменений.

Если вы не знаете, как запустить CGI-программу без анализа заголовков, обратитесь к документации по Web-серверу или проконсультируйтесь у системного администратора.

Маленький пример: обновление часов

В качестве первого примера работы сервера в режиме выталкивания страниц напишем простую программу, обновляющую часы на Web-странице. Часы работают за счет того, что сервер примерно каждые 5 секунд будет отправлять броузеру очередную страницу с новым временем. Web-сервер будет продолжать выталкивать страницы до тех пор, пока броузер не "уйдет" с этой страницы или пользователь не щелкнет на кнопке броузера Остановить (Stop), чтобы прекратить процесс загрузки страницы.

В модуле CGI предусмотрен набор функций, разработанных для упрощения процесса выталкивания страниц сервером. Web-страницы, обновляемые сервером, также называются *составными документами (multipart documents)*.

В листинге 23.1 представлен исходный код CGI-программы вывода времени сервером. Вам нужно ввести этот код и сохранить его в файле с таким именем, чтобы Web-сервер мог выполнить его как CGI-программу без анализа заголовков (см. описание в предыдущем разделе).

Листинг 23.1. CGI-программа вывода времени сервером

```
1:  #!/usr/bin/perl -w
2:
3:  use strict;
4:  use CGI qw(:push -nph);
5:
6:  $|=1;  # Разрешена автоматическая очистка буфера
7:
8:  print multipart_init;
9:  while(1) {
10:     print multipart_start;
11:     print "Местное время <H1>", scalar(localtime), "</H1>\n";
12:     print multipart_end;
13:     sleep 5;
14: }
```

Проведем анализ программы.

- *Строка 4.* При загрузке CGI-модуля необходимо указать, что он будет выполняться в режиме выталкивания страниц, поэтому данная строка содержит директиву :push для модуля CGI. Кроме того, при написании сценариев без анализа заголовков необходимо уведомить об этом модуль CGI с помощью аргумента -nph.

- *Строка 8.* Функция `multipart_init` информирует броузер о том, что далее будет следовать составная Web-страница. Эта функция используется вместо функции header в обычных Web-страницах.
- *Строка 9.* С помощью выражения `while(1)` создается цикл while, который будет повторяться *бесконечно долго.*
- *Строка 10.* Функция `multipart_start` отмечает начало обновляемой страницы. Если страница уже отображается, эта строка заставляет броузер очистить ее и ожидать приема нового содержимого.
- *Строка 11.* Эта строка представляет собой содержимое составной страницы. Как упоминалось на 4-м занятии, "Укладка строительных блоков: списки и массивы", функция localtime в скалярном контексте выводит время в формате "Sun Sep 5 15:15:30 1999".
- *Строка 12.* Функция multipart_end отмечает конец обновляемой страницы. За этой строкой может следовать только другая функция `multipart_start` или конец программы.

  Обратите внимание на то, что функции raultipart_init и raultipart_end находятся в теле цикла while. Этот цикл снова и снова отображает одну и ту же страницу: в ней меняется лишь значение времени суток.

Еще один пример: анимация графического изображения

В следующем примере (листинг 23.2), который во многом напоминает предыдущий, отображается набор изображений из каталога /images. Они отображаются поочередно с использованием режима выталкивания страниц. Вначале выполняется чтение всех файлов из каталога, а затем их содержимое поочередно посылается броузеру в виде набора "вытолкнутых" страниц.

Листинг 23.2. Графическая анимация с использованием режима выталкивания страниц

```
#!/usr/bin/perl -w

use strict;
use CGI qw(:push -nph);
my($imagedir, @JPEGS);
\ Замените имя каталога на реальное
$imagedir="/web/Clinton Test_Area/images";
opendir(ID, $imagedir) J[ die "Ошибка при открытии $imagedir: $1";
@JPEGS=sort grep(/\.jpg$/, readdir ID);
closedir{ID};

$[=1; # Разрешена автоматическая очистка буфера

print multipart_init;
foreach my $image (@JPEGS) {
   print multipart_start(-type => 'image/jpeg');

   open(IMAGE, "$imagedir/$image") ||
       die "Ошибка при открытии $image: $!";
```

```
19:     binmode(STDOUT); binmode(IMAGE);  # Только для Windows NT/95/98
20:     print <IMAGE>;
21:     close(IMAGE);
22:     print multipart_end;
23:     sleep 5;
24: }
```

Программа, представленная в листинге 23.2, большей частью напоминает программу вывода изображения дня и ночи, приведенную на 20-м занятии, "Работа с HTML-кодом и CGI-программами", и пример, содержащийся в листинге 23.1.

Важным элементом, на который стоит обратить внимание, является строка 16: `multipart_start(-type => 'image/jpeg')`. Она означает, что вместо вывода обычного текста или документа HTML на последовательных Web-страницах CGI-программа будет выводить изображения в формате JPEG. Для выполнения анимации CGI-программа должна либо напрямую выводить JPEG-файлы, либо выводить HTML-код, содержащий дескрипторы <IMG SRC>.

Сравнение с методом вытаскивания страниц клиентом

Другой метод последовательной загрузки Web-страниц предполагает участие клиента. В соответствии с этим методом в HTML-код помещаются дескрипторы, указывающие броузеру на необходимость перезагрузки той же страницы (или страницы с другим URL), в которых указывается интервал времени. Например, приведенный ниже HTML-дескриптор, помещенный в раздел <HEAD> Web-страницы, должен заставить броузер загрузить страницу `http://foo.bar.com` через 6 секунд.

```
<META HTTP-EQUIV="refresh" CONTENT="6;http://foo.bar.com">
```

Модуль CGI косвенно поддерживает директивы загрузки страниц по запросу клиента.

При выводе заголовка Web-страницы с помощью аргумента -Refresh функции header модуля CGI можно указать, что данная страница должна быть перезагружена или на ее место будет загружена другая страница:

```
print header(-Refresh => '6;URL=http://foo.bar.com');
```

В действительности участие клиента заключается в загрузке новой страницы между "обновлениями". Это означает, что если у вас есть страницы, которые необходимо отобразить в определенной последовательности (как при показе слайдов), вам нужно с помощью файлов cookie или параметров, встроенных в URL, отслеживать эту последовательность, т.е. то, какая страница должна быть отображена следующей. В период между обновлениями между сервером и клиентом должно быть установлено новое соединение, и Web-сервер должен запустить новую Perl-программу для каждого обновления. А это, скорее всего, означает, что обновления не могут происходить слишком быстро.

Ниже перечислены основные проблемы, возникающие при использовании любого из описанных выше методов загрузки некоторой последовательности страниц (путем выталкивания сервером или вытаскивания клиентом).

- Некоторые броузеры не поддерживают режим выталкивания страниц "силами" сервера, в частности броузер Internet Explorer.
- Некоторые броузеры не поддерживают режим вытаскивания страниц "силами" клиента.

Какой бы из методов вы ни использовали при написании **CGI-программ**, они в чем-то окажутся несовместимыми с тем или иным броузером. Тем не менее вам придется принимать решение по выбору "наименьшего зла" и писать соответствующий программный код.

Счетчики посещений

На Web-страницах иногда можно увидеть нечто, именуемое счетчиком щелчков или индикатором числа посещений. Предположительно он отражает, сколько раз посетили данную Web-страницу. Пример такого индикатора показан на рис. 23.3.

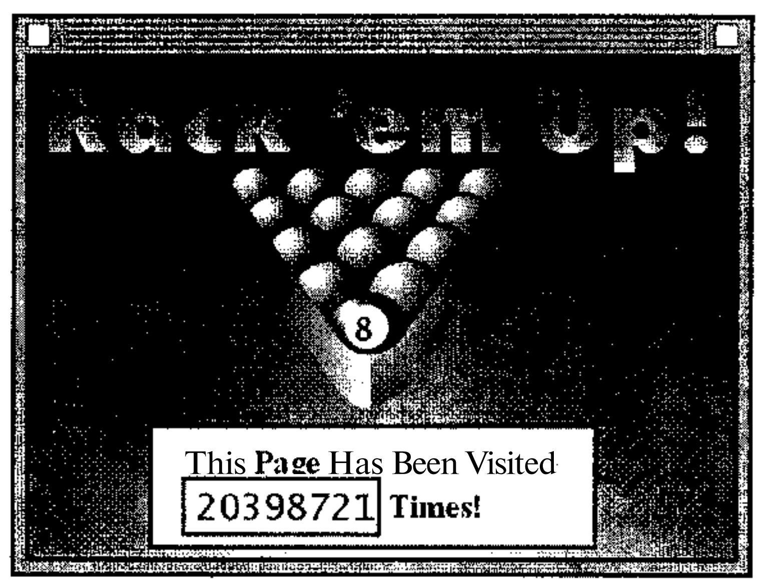


Рис. 23.3. Пример индикатора посещений Web-страницы

Показания счетчиков посещений связаны с множеством проблем. Прежде всего, что означает это число? Можно предположить, что большое значение этого числа "щелчков" говорит о высокой популярности данной страницы. Но разве популярность страницы означает, что она так уж хороша? Совсем необязательно. Посетив какую-нибудь Web-страницу, вы либо находите нужную информацию, либо нет. Качество страницы — в ее значимости именно для вас, а не для других.

По сути, счетчик посещений — это своего рода конкурс красоты с незрячими членами жюри. Представленное счетчиком число совсем не означает число людей, посетивших вашу страницу. В лучшем случае — это просто неточная оценка. Почему же эти счетчики так неточны? Вероятно, на это есть несколько причин.

Прежде всего, следует сказать, что нет такого правила, которое бы предписывало, чтобы счетчик посещений начинал отсчет с нуля. Когда вам выдавали чековую книжку, то номер первого чека разве был равен #1? По всей вероятности, при заказе чековой книжки вам разрешалось выбрать начальный номер чека. И если у вас есть некоторый опыт в общении с банковскими служащими, вы выберете в качестве начального номер побольше, чтобы тем самым создать впечатление, что у вас довольно продолжительные отношения с этим банком. Маленький номер может заставить клерка дважды проверить ваш идентификатор и не исключено, что он вообще откажется принимать чек. Администраторы Web-узлов часто устанавливают начальные значения счетчиков посещений большими, чтобы их Web-узлы выглядели более "популярными", чем они есть на самом деле.

Вторая проблема, связанная со счетчиками посещений, — это Web-роботы, которые также называются пауками, червями и пр. Они представляют собой автоматические процессы поиска данных в Web (иногда просто с целью отыскать некоторую справочную информацию, а иногда для построения индексов интересных Web-узлов). Вы когда-нибудь интересовались, как AltaVista, Google или HotBot строят свои индексы? Они "просматривают" Web, посещают "нужные" страницы, что в конце концов приводит к "взлету" показаний счетчиков на незаслуженную высоту.

Следующая проблема связана с кнопкой Refresh (Обновить) Web-броузеров. При каждом "обновлении" счетчик посещений делает новую "зарубку". Тогда выходит, что если щелчок на кнопке перезагрузки увеличивает показания счетчика, то эти показания на самом деле не означают количество посетителей вашего узла, не так ли?

Последняя и самая важная проблема связана с кэшированием. На 17-м занятии, "Введение в CGI", была приведена схема подключения броузера к Web-серверу. В ней не учтены некоторые важные детали, которые показаны на рис. 23.4.

Если Web-броузер размещен внутри домена такого крупного провайдера услуг Internet (ISP), как aol.com или home.com с многомиллионным отрядом абонентов, то такой ISP использует кэширующее устройство, именуемое *proxy-сервером.* Это кэширующее устройство можно рассматривать как посредника между Web-броузером и Web-сервером. После щелчка на гиперссылке некоторой Web-страницы этот запрос отправляется proxy-серверу, который "достает" эту страницу для вас из Internet и отправляет ее обратно вашему броузеру. При этом proxy-сервер сохраняет копию этой страницы для "себя" (рис. 23.5). Если кто-нибудь еще в том же домене запросит ту же страницу, proxy-сервер не станет утруждать себя повторной ее загрузкой из Internet, а воспользуется сохраненной (на всякий случай) копией.

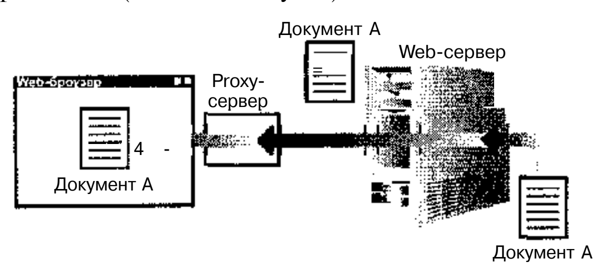


Рис. 23.4. Proxy-сервер "достает" страницу для броузера

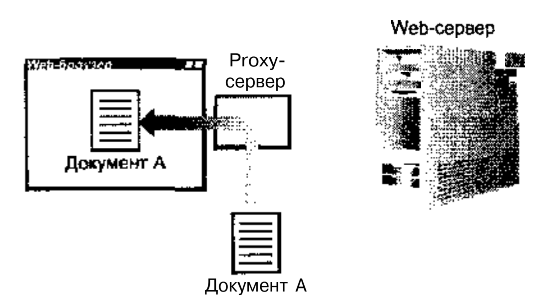


Рис. 23,5. Proxy-сервер считывает страницу из своего кэша

При сохранении на proxy-сервере копии страницы искусственно уменьшается количество посещений, отображаемых счетчиком. При кэшировании функция `remote_host` часто будет возвращать одно и то же значение, даже несмотря на то, что страница запрашивается различными пользователями.



Пользователи Web, находящиеся в больших организациях и университетах, часто защищены системами-брандмауэрами (firewall), которые действуют подобно proxy-серверам. Каждый запрос страницы, исходящий от одного из этих узлов, скорее всего не будет учтен в счетчике посещений, поскольку он перехватывается proxy-сервером.

А теперь, **собственно,** перейдем к счетчику посещений

Если, прочитав предыдущий раздел, вы еще не отложили эту книгу в сторону, то можно предположить, что вас интересует создание такого счетчика посещений для своей Web-страницы. Есть два основных типа счетчиков посещений для Web-страниц: текстовый и графический. Сначала рассмотрим текстовый счетчик, а затем — графический, после чего обсудим возможности улучшения "моделей" счетчиков.

Для обеспечения работоспособности счетчика посещений используйте возможности серверных включений (server-side include — SSI), с которыми вы познакомились на 20-м занятии, "Работа с HTML-кодом и CGI-программами". Если, например, CGI-программу организации работы такого счетчика назвать `hits.cgi`, то с помощью дескрипторов SSI ее можно включить в любую страницу:

```
<!--#exec cgi="/cgi-bin/hits.cgi"-->
```

Исходный код CGI-программы счетчика посещений представлен в листинге 23.3.

Листинг 23.3. Программа счетчика посещений Web-страницы

```
#!/usr/bin/perl -w

use strict;
use Fcntl qw(:flock);
use CGI qw(:all);

my $semaphore_file='/tmp/webcount_lock';
my $counterfile='/web/httpd/countfile';
sub get_lock {
    open(SEM, ">$semaphore_file")
        || die "Не удается создать семафор: $!";
   flock(SEM, LOCK_EX) || die "Блокировка не удалась: $!";
}
# Функция снятия блокировки
sub release_lock {
    close(SEM);
}
get_lock();  # Блокировка и ожидание.
my $hits=0;
if ( open(CF, $counterfile) ) {
    $hits=<CF>;
    close(CF);
}
$hits++; # Увеличение счетчика посещений на 1.
print header;
print "Вы стали $hits посетителем";

```

```
28:  open(CF, ">$counterfile") ||
        die "Ошибка при открытии $counterfile: $!";
29:  print CF $hits;
30:  close(CF);
31:
32:  release lock();  # Снимаем блокировку
```

 Проведем анализ программы.

- *Строка 18:* Без блокировки не обойтись, поскольку с файлом счетчика посещений могут выполнять операции чтения и записи несколько процессов одновременно.
- *Строки 20–23.* Считывается содержимое файла, имя которого указано в переменной `$counterfile`. Пока это и есть количество посещений.
- *Строки 28–30.* Содержимое счетчика посещений снова записывается в файл, имя которого указано в переменной $counterfile.
- *Строка 32.* Снимается блокировка.

Большая часть программы, представленной в листинге 23.3, не должна отпугивать вас неизвестными элементами. Однако обратите внимание на выполнение операции блокировки файла и примеры с использованием блокировок, описанные на 15-м занятии, "Обработка данных в Perl".

Блокировку файлов необходимо выполнить на случай, если два пользователя попытаются практически одновременно загрузить вашу Web-страницу. Если операции чтения и записи файла счетчика посещений Web-страницы не синхронизировать, то показания этого счетчика могут увеличиваться или слишком быстро, или слишком медленно, а то и вообще может произойти порча файла. В этом случае вряд ли можно говорить о приемлемой точности показаний счетчика.

Графический счетчик посещений

Чтобы внести изюминку в оформление счетчика посещений, можно воспользоваться различными способами. Остановимся на трех. Первый предполагает составление графического изображения, представляющего все возможные значения счетчика, и его отображение при необходимости. Этот метод отличается большими временными затратами при увеличении числа посетителей Web-узла.

Второй метод состоит в использовании CGI-программы на языке Perl, которая бы сама генерировала нужные графические изображения для счетчика посещений. Модуль GD, доступный в CPAN, как раз предназначен для создания графических изображений с использованием Perl-программ. Воспользуйтесь им, чтобы внести в свой счетчик элемент оригинальности. К сожалению, подробное знакомство с модулем GD выходит за рамки этой книги.

Проще всего создать 10 изображений, представляющих цифры от 0 до 9. Затем при увеличении показаний счетчика ваша программа может выводить HTML-код с дескрипторами <IMG>, которые поместят эти цифры на нужное место (рис. 23.6). Но вам, конечно, придется создать изображения, представляющие эти цифры. CGI-программа на языке Perl, представленная в листинге 23.4, написана в предположении, что файлы изображений имеют следующие имена: digit_0.jpg, digit_1.jpg и т.д., вплоть до файла digit_9.jpg.

```
<IMG SRC="digit_1.jpg">
<IMG SRC="digit_0.jpg">
<IMG SRC="digit_3.jpg">     ———→  | 1 | 0 | 3 | 4 |
<IMG SRC="digit_4.jpg">
```

Рис. 23.6. Результат работы графического счетчика посещений

Для обеспечения работоспособности счетчика запускайте CGI-программу в режиме серверного включения, как было описано на 20-м занятии, "Работа с HTML-кодом и CGI-программами". Если, например, CGI-программу такого счетчика назвать `graphical_hits.cgi`, то с помощью дескрипторов SSI ее можно включить в любую страницу:

```
<!--#exec cgi="/cgi-bin/graphical_hits.cgi"-->
```

Исходный код **графического счетчика** посещений Web-страницы представлен в листинге 23.4.

Листинг 23.4. Графический счетчик посещений Web-страницы

```
#!/usr/bin/perl -w

use strict;
use Fcntl qw(:flock);
use CGI qw(:all);

my $lockfile='/tmp/webcount_lock';
my $counterfile='/web/httpd/countfile';
my $image_url='http://www.server.com/images';

sub get_lock {
  open(SEM, ">$lockfile")
      || die "Не удается создать семафор: $!";
   flock(SEM, LOCK_EX) || die "Блокировка не удалась: $!";
}
sub release_lock {
   close(SEM);
}
get_lock(); # Выполнение блокировки и ожидание.
my $hits=0;
if ( open(CF, $counterfile) ) {
     $hits=<CF>;
     close(CF);
}
$hits++;

open(CF, ">$counterfile") ||
     die "Ошибка при открытии $counterfile: $!";
print CF $hits;
close(CF);
release_lock(); # Снятие блокировки

# Теперь создаем дескрипторы <IMG>.
print header;
foreach my $digit (split(//, $hits)) {
   print "<IMG SRC=$image_url/digit_$digit.jpg>";
}
```

 Листинг 23.4 по сути повторяет листинг 23.3, но содержит небольшие отличия.

- *Строка 9.* Здесь в переменной `$image_url` содержится базовый URL для изображений цифр. Однако следует помнить, что это должен быть URL, которым для загрузки изображений воспользуется броузер, а не полный путь к файлам изображений, находящихся на локальном диске.
- *Строки 34–35.* Число в счетчике посещений — `$hits` - дробится на отдельные символы, и каждая цифра по очереди присваивается переменной `$digit`. Затем для каждой цифры создаются дескрипторы <IMG>.

Резюме

На этом занятии вы познакомились с двумя методами реализации анимации на Web-страницах. Вы можете воспользоваться методом выталкивания страниц сервером (server push), чтобы заставить броузеры постоянно обновлять Web-страницы. Если этот метод вам не подходит или он не поддерживается определенным типом броузера, то для получения аналогичных эффектов можно использовать метод вытаскивания страниц со стороны клиента (client pull). Кроме того, вы узнали о счетчиках посещений Web-страниц, а также о причинах их вопиющей неточности.

Вопросы и ответы

Почему не работает метод выталкивания страниц сервером?

Данный метод может не работать по различным причинам. Во-первых, push-технологию должен поддерживать броузер. Во-вторых, Web-сервер должен запускать CGI-программы без анализа заголовков. И, наконец, ваша CGI-программа должна работать корректно. При запуске CGI-программы из командной строки убедитесь, что результаты выводятся через регулярные интервалы времени и при этом верны.

Если счетчики посещений настолько плохи, то существует ли надежный способ оценить посещаемость Web-узла?

Практически нет. Анализ обращений к серверу по системному журналу дает такие же неточные результаты, как и использование счетчика посещений. Можно подсчитать количество обращений к конкретной странице с помощью гиперссылки с перенаправлением (см. материал 20-го занятия, "Работа с HTML-кодом и CGI-программами"). Другой вариант — вынудить посетителей физически заполнять некоторую форму. Использование метода POST при обработке данных HTML-форм является единственным способом, позволяющим со стопроцентной гарантией избежать кэширования страниц proxy-сервером.

Семинар

Контрольные вопросы

1. Какие функции из модуля CGI нужно использовать для реализации метода выталкивания страниц сервером?

   а) multipart_start и `multipart_end`.

б) multipart_init, multipart_start и multipart_end.

в) push_start и push_end.

2. Все броузеры поддерживают метод выкачки страниц со стороны клиента, поскольку он является частью HTML-стандарта.

   а) истина;

   б) ложь.

3. Какие виды Web-страниц proxy-серверы гарантированно не кэшируют?

   а) данные HTML-формы, которая использует для их обработки метод POST;

   б) содержимое страницы, выталкиваемой сервером;

   в) результаты работы любой CGI-программы.

Ответы

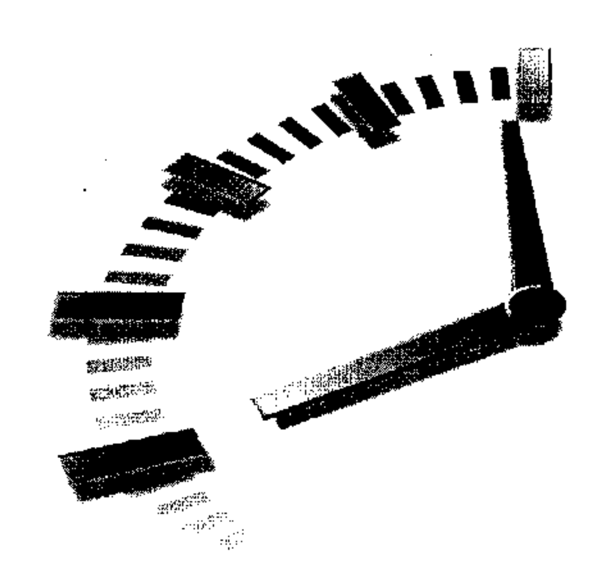
1. Правильный ответ — вариант б). Функция multipart_init используется для подготовки броузера к приему составных Web-страниц. Функции multipart_start и multipart_end отмечают начало и окончание каждой такой страницы.

2. Правильный ответ — вариант б). Конечно же, ответ — ложь. Как утверждает стандарт, дескрипторы <META> могут игнорироваться броузерами. Кроме того, использование параметра -Refresh в функции header не гарантирует, что броузер по требованию будет перезагружать страницу; броузер также имеет возможность игнорировать эту директиву.

3. Правильный ответ — вариант а). Ответ был разъяснен в разделе "Вопросы и ответы".

Упражнения

- Модифицируйте CGI-программу счетчика посещений в листинге 23.3 (или 23.4), чтобы в ней поддерживались разные счетчики для различных типов броузеров. Для этого вам нужно иметь отдельные файлы для каждого типа броузера. Не забудьте о дополнительном файле для тех броузеров, которые программа сможет идентифицировать.



24-й час

Создание интерактивного Web-сервера

Если вы используете **CGI-программы** на своем Web-сервере для того, чтобы привлечь людей и предоставить им интересное место для посещения, то вам нужно сделать нечто большее, чем просто создать счетчик числа посещений.

Самыми интересными серверами в Web являются те, содержимое которых постоянно обновляется. Если на вашей Web-странице информация не меняется, то у людей нет никакой причины когда-либо возвращаться к ней снова. Посетив пару раз вашу страницу, они поймут, что ее содержимое почти не изменилось и снова посещать ее не стоит.

Есть еще кое-что, заставляющее людей возвращаться к Web-серверу, — это их участие тем или иным способом в его работе. Люди, составляющие определенную группу, любят поговорить о себе. Одно из свойств человеческой натуры заключается в том, что люди хотят принадлежать к некой группе и чувствовать себя участвующими в ее жизни и работе.

Программы, описанные на этом занятии, предоставят возможность сделать и то, и другое. Первая программа позволяет отсканировать информацию из другого источника в Internet, переформатировать ее и представить на вашем сервере со ссылкой на первоначальный источник информации. Вторая программа позволяет посетителям сервера участвовать в опросе.

Основные темы этого занятия.

- Как заимствовать информацию из других Web-серверов.
- Как создать Web-сервер для проведения интерактивного опроса.

Заимствование информации

Вы, наверное, видели Web-серверы, на страницах которых были достаточно новые данные котировок ценных бумаг, заголовки последних новостей или сведения о том, с каким счетом закончилась та или иная спортивная игра. Причем иногда человек, администрирующий данный Web-узел, не имеет никакого отношения к первоначальным источникам информации.

В таких случаях обычно происходит следующее: программа, работающая на компьютере, на котором находится рассматриваемая вами Web-страница, время от времени связывается с первоначальным источником информации и "вытягивает" из него данные. Затем эта информация переформатируется и отображается на заданной странице. На рис. 24.1 показано, как выглядит этот процесс.

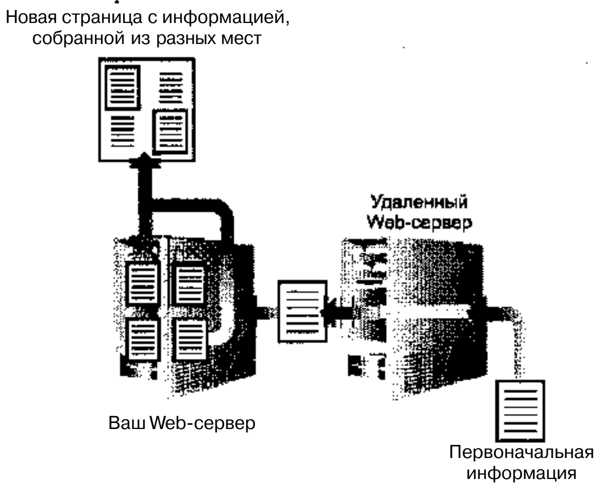


Рис. 24.1. Извлечение Web-страницы, ее переформатирование и отображение на другой странице

Таким образом, вы видите, что ваш Web-*сервер* с помощью CGI-программ превратился в *Web-клиента.* Если сервер может сам извлекать страницы, то он сможет также объединить их вместе и вновь представить информацию уже иным способом.

Важный момент: не играйте с огнем

Не спешите сразу переходить к заимствованию информации. Для начала следует запомнить несколько важных вещей. Во-первых, если информация, представленная на вашей странице, не является вашей собственностью (она получена из другого Web-сервера или чей-то базы данных), то, вероятно, она защищена авторским правом. Заимствование информации на одном Web-сервере для представления ее на другом может привести к серьезным проблемам с законом. Нарушение закона об авторском праве может повлечь за собой закрытие вашего Web-сервера, наложение санкций на вашего провайдера, а вас могут оштрафовать, заключить в тюрьму и возбудить уголовное дело.

Все это возможные следствия отсутствия культуры работы в Internet.

Прежде чем использовать информацию из другого источника на своем Web-сервере, всегда спрашивайте разрешение на это. Большинство владельцев Web-серверов разрешат вам отображать содержимое их страниц, но могут попросить вас соблюдать некоторые правила и выполнить несколько ответных шагов, которые перечислены ниже.

- Вы указываете источник информации — возможно, с помощью баннера, гиперссылок или текста.
- Четко формулируете, кому принадлежит авторское право на представляемую информацию, и указываете, что она используется с разрешения автора.

- Возможно, вам не разрешат создать "глубокую связь" с авторским Web-сервером, т.е. связываться со страницами, которые находятся на несколько уровней глубже главной страницы. Скорее всего, владельцы сервера предпочтут, чтобы вы создали ссылку только на страницу самого верхнего уровня.
- Возможно, вам разрешат обновлять "вытягиваемую" с сервера информацию только время от времени. Некрасиво перегружать трафик на чужом сервере только ради того, чтобы улучшить впечатление от своего.

Владельцы Web-сервера Slashdot.Org, девиз которых звучит "Новости для технарей", а информация рассчитана на пользователей с техническим образованием, позвонили мне использовать их сервер при написании книги, чтобы на примерах продемонстрировать возможности языка Perl. Поэтому, прежде чем реализовать коды программ из описанных ниже примеров на собственных Web-страницах, вы должны спросить разрешения у их авторов. Подробные сведения о том, как с ними связаться, можно найти на их Web-сервере и в разделе FAQ по адресу http://www.slashdot.org.

Пример: „вытягивание" заголовков

Схема отображения заголовков сервера Slashdot.Org на вашей Web-странице выглядит примерно так.

1. CGI-программа headlines.cgi запускается через анализируемую сервером *Web-страницу*.
2. Затем CGI-программа проверяет, самый ли свежий вариант заголовков находится на локальном диске. Если да, то она использует его. В противном случае программа "вытягивает" новые заголовки с Web-сервера Slashdot.Org.
3. После этого CGI-программа анализирует файл заголовков и отображает их на своей Web-странице.

Для "вытягивания" Web-страницы или любого другого содержимого какого-либо Web-сервера вам нужен модуль LWP::Simple, который не входит в стандартную поставку Perl. Модуль LWP позволяет "вытягивать" из Internet все виды информации: Web-страницы, файлы через протокол FTP, статьи из групп новостей и т.д.



> Модуль LWP::Simple является составной частью пакета под именем libwww-perl. В этом пакете содержатся модули для "вытягивания" Web-страниц, анализаторы HTML-кода и URL, средства навигации по Web-серверу и многое другое. Эти модули дают вам колоссальные преимущества, так что имеет смысл потратить время на их установку. Пакет libwww-perl находится на компакт-диске, прилагаемом к данной книге.

Установив модуль LWP::Simple, вы сможете "вытянуть" Web-страницу следующим образом:

```
use LWP::Simple qw(get);
$content=get("http://www.slashdot.org");
```

Теперь переменная $content содержит текст Web-страницы, находящейся по указанному URL. Просто, не правда ли?

Программа "вытягивания" заголовков сервера Slashdot и их последующего отображения приведена в листингах 24.1—24.3. Чтобы использовать данную программу, нужно запустить ее непосредственно из командной строки или из анализируемой сервером Web-страницы.

Листинг 24.1. Программа "вытягивания" заголовков сервера Slashdot: часть I

```
1:  #!/usr/bin/perl -w
2:
3:  use strict;
4:  use Fcntl qw(:flock);
5:  use LWP::Simple qw(get);
6:  use CGI qw(:all);
7:
8:  my $url="http://slashdot.org/slashdot.xml";
9:  my $cache="/tmp/slashcache";
10: my $lockfile="/tmp/slashlock";
11: sub get_lock {
12:     open(SEM, ">$lockfile")
13:        || die "Ошибка при создании семафора: $1";
14:    flock(SEM, LOCK_EX) || die "Ошибка при выполнении блокировки: $!";
15: }
16: sub release_lock {
17:    close(SEM);
18: }
```

Проведем анализ программы.

- *Строки 3—6.* Для написания этой программы понадобится несколько дополнительных модулей. Следует использовать модуль Fcntl, поскольку нужно, чтобы программу одновременно мог запускать только один пользователь. Вам понадобится также модуль LWP::Simple, точнее его функция get, чтобы "вытянуть" заголовки из Web-сервера Slashdot. И, конечно, вам необходим модуль CGI, поскольку вы собираетесь создать CGI-программу.

- *Строка 8.* Здесь указывается URL файла, в котором находятся заголовки. Формат этого файла выглядит примерно так:

```
<story>
    <title>Ask Slashdot: Internet Voting?</title>
    <url>http://slashdot.org/askslashdot/99/09/05/1732249.shtml</url>
    <time>1999-09-05 21:34:36</time>
    <author>Cliff</author>
     :
```

  Каждый элемент заключен между дескрипторами <story>. Для написания этого файла использован более современный вариант языка разметки под названием *XML*. Как вы увидите позже, это позволяет значительно упростить его обработку Perl-программой.

- *Строка 9.* Переменная $cache содержит имя файла, в котором вы собираетесь временно хранить заголовки сервера Slashdot. Благодаря этому файлу каждый раз при вызове программы вам не нужно опрашивать сервер Slashdot, чтобы получить информацию, поскольку у вас есть ее локальная копия.

- *Строки 11–18.* Подпрограммы `get_lock()` и `release_lock()` должны быть вам хорошо знакомы, так как вы уже встречались с ними на трех занятиях. Эти подпрограммы нужны вам, потому что файл, имя которого хранит переменная $cache, может обновляться не более чем одной программой одновременно, и, следовательно, его нужно заблокировать.

Листинг 24.2. Программа "вытягивания" заголовков сервера Slashdot: часть II

```
19:
20: print header;
21: # Если с момента обновления кэш-памяти прошло больше часа, обновим ее
22: get_lock();
23: if ( (not -e $cache ) or ( (-M $cache) > .04)) {
24:     my $doc=get($url);
25:     if (defined $doc) {
26:             open(CF, ">$cache") || die "Ошибка записи в кэш: $!";
27:             print CF $doc;
28:             close(CF);
29:     }
30: }
31: release_lock();
32:
```

- *Строка 23.* Определяет следующее: если файл в кэш-памяти отсутствует или если ее содержимое хранится уже более 60 минут, то следует обновить информацию. Функция `-M` в Perl возвращает время модификации файла с момента запуска Perl-программы. Это время выражается в *дробных долях дня.* Так, если файлу один день, то функция -м возвращает значение 1; если файлу 6 часов, то `-M` возвращает значение 0.25 (четверть дня); если файлу всего один час, то `-M` возвращает значение `0.0416666` (приблизительно 1/24).

- *Строка 24.* С помощью функции get модуля `LWP::Simple` осуществляется выборка заголовков по заданному URL, как было описано выше. В следующих нескольких строках извлеченный документ, который хранится в переменной $doc, записывается в кэш-файл. Если выполнение функции get завершилось неудачей, она возвращает значение `undef`, которое проверяется в строке 25.

Обратите внимание, что вызовы функций `get_lock()` и release_lock() находятся вне "сферы влияния" оператора if. Этот момент очень **важен**. Если один экземпляр **CGI-программы** обновляет кэш-файл, то вам совсем не нужно, чтобы другая программа одновременно проверяла, существует ли кэш-файл и когда он модифицировался последний раз.

Листинг 24.3. Программа "вытягивания" заголовков сервера Slashdot: часть III

```
33: print "<H2>Slashdot.Org's Headlines as of ",
34:     scalar(gmtime((stat $cache)[9])),
35:     "GMT </H2>Updated Hourly!<P>";
36:
37: open(CF, $cache) || die "Ошибка при открытии кэш-файла: $!";
38: my($title, $link);
39: while(<CF>) {
40:     if (m,<title>(.*)</title>,) {
```

```
41:             $title=$1;
42:     }
43:     if (m,<url>(.*)</url>,) {
44:             $link=$1;
45:             print qq{<A HREF="$link">$title</A><BR>\n};
46:
47:     >
48: }
49: print "Авторское право принадлежит Slashdot.Org;",
        " материал используется с разрешения данной организации.";
50: close(CF);
```

Последняя часть этой программы — самая простая.

- *Строки 33–35.* Здесь отображается вводная информация и время последнего обновления кэш-файла.
- *Строка 34.* Здесь содержится некий хитроумный прием, позвольте мне его объяснить. Сначала функция stat получает информацию о файле, имя которого хранится в переменной $cache, и возвращает ее в виде списка. Далее извлекается 9-й элемент (время последней модификации) этого списка. И, наконец, для этого времени вызывается функция localtime, которая в скалярном контексте возвращает хорошо отформатированную строку.
- *Строки 40–43.* Извлекаются разделы <title> и <url> файла заголовков сервера Slashdot. После анализа регулярного выражения соответствующие части заглавия и URL сохраняются в переменной $1, а затем присваиваются переменным $title и $link соответственно.
- *Строка 45.* Поскольку элемент <url> всегда идет после элемента <title>, то при обработке элемента <url> значение обеих переменных $title и $link уже может быть выведено в выходной поток.

В общем случае эти регулярные выражения *не* следует использовать для обработки HTML-файлов. Они используются здесь, потому что XML-файл заголовков сервера Slashdot тщательно отформатирован так, что в каждой строке содержится ровно один XML-элемент. Если вам покажется, что формат этого файла изменился и эта программа не может обработать его, то следует обратиться к списку часто задаваемых вопросов сервера Slashdot, чтобы узнать, что случилось.

После того как вы запустите эту программу, результат (т.е. ее выходные данные) будет выглядеть примерно так, как на рис. 24.2.

Разумеется, вам придется приложить все свое искусство форматирования HTML-документа, чтобы придать этим данным более привлекательный вид.

Каковы результаты опроса?

Каждый хочет быть *кем-то.* Каждый хочет знать, что с его мнением считаются, и — что тоже важно — каждый хочет знать, насколько его мнение соответствует мнениям других. Для этого и проводятся опросы.

В следующем упражнении мы рассмотрим небольшую программу создания опроса, а затем программу печати результатов этого опроса. Данные опроса сохраняются в текстовом файле: сначала идет сам вопрос, а затем — несколько вариантов ответов.

Файл помешается в каталог на Web-сервере и к его имени добавляется расширение `.txt`. Содержимое файла выглядит примерно так, как показано ниже, причем в нем не должно быть никаких других знаков препинания или пустых строк.

```
Ваше любимое домашнее животное:
Собака
Кошка
Рыбки
Они меня раздражают
```

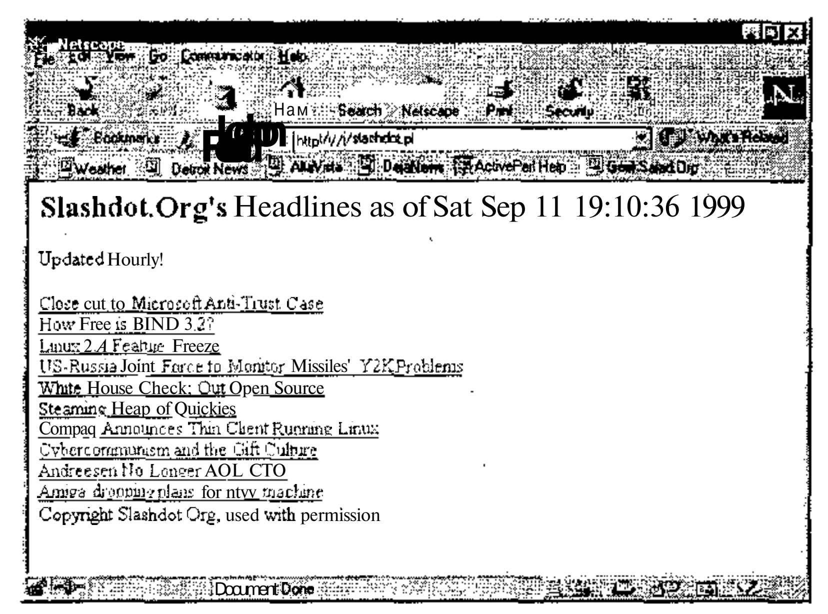


Рис. 24.2. Выходные данные программы `slashdot.cgi`

Первая программа ищет в этом каталоге файл с расширением `.txt`, выбирая последний, если там есть несколько файлов, и отображает вопросы в виде анкеты, как показано на рис. 24.3.

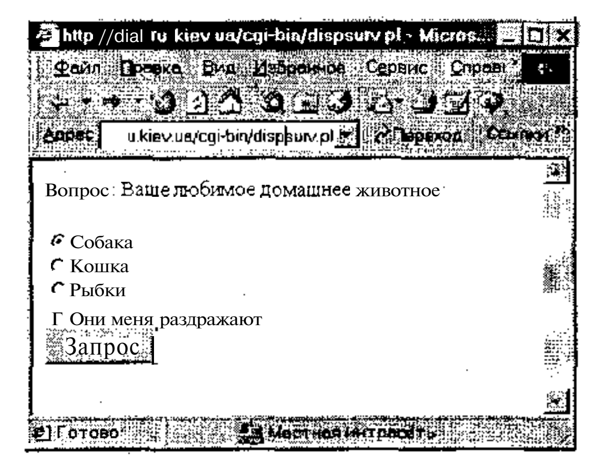


Рис. 24.3. Форма для проведения опроса

Преимущество использования простого текстового файла состоит в том, что CGI-программа может его использовать для отображения вопросов, а затем и ответов. Если вы хотите составить новый опрос, то просто поместите еще один .txt-файл в тот же каталог, и тогда **CGI-программа** начнет использовать его автоматически. Все операции выполняются автоматически без вашего участия.

Когда пользователь выбирает вариант ответа и щелкает на кнопке подачи запроса (Submit), запускается вторая CGI-программа, которая записывает данные опроса в файл, находящийся в том же каталоге, что и вопросы. Если файл вопросов называется foo.txt, то ответы сохраняются в файле с именем `foo.answer`. Когда CGI-программа закончит запись ответа, она перечитывает все ответы и отображает результаты.

Часть I: постановка вопросов

Ставить вопросы в данном опросе совсем несложно. Самое сложное — это просмотреть каталог, где хранится информация об опросе, и найти в нем последний файл с расширением `.txt`. На самом деле, поскольку данный файл нужно найти обеим программам — задающей вопрос и записывающей результаты, — имеет смысл написать этот фрагмент в виде повторно используемой функции, которую можно применить и в том, и в другом случае.

Листинг 24.4. Отображение опроса, часть I

```
#!/usr/bin/perl -w

use strict;
use CGI qw(:all);
my($survey_dir);
$survey_dir="/web/htdocs/poll";

sub find_last_file {
        my($type)=@_;
        my(@files, $last_file);
        It Откроем каталог и найдем последний файл
        # нужного типа.
        opendir{SD, $survey_dir) ||
            die "Ошибка при открытии $survey_dir: $!";
        @files=reverse sort grep(/\.$type$/, readdir SD);
        closedir(SD);
        $last_file=$files[$#files];
        return($last_file);
}

sub get_file_contents {
        my($type)=@_;
        my(@answers, $last_file);

        $last_file=find_last_file($type);

        return if (not defined $last_file);
        # Откроем этот файл и прочитаем его содержимое.
        open(QF, "$survey_dir/$last_file")
```

```
29:             || die "Ошибка при открытии $last_file: $!";
30:         @answers=<QF>;
31:         close(QF);
32:         chomp @answers;   # Удалить символы новой строки
33:         return(@answers);
34: }
```

Проведем анализ программы.

- *Строка 6.* В переменной $survey_dir задается имя каталога, где должны находиться файлы опроса. Чтобы создать новый опрос, вы можете просто поместить текстовый файл с расширением `.txt` в этот каталог, как было описано в начале данного раздела. Этот каталог должен быть доступным для записи процессам Web-сервера. Доступность для записи обычно означает наличие у пользователя прав доступа 0755 в системе UNIX или разрешение на запись в данный каталог для учетной записи guest в системе Windows NT.

- *Строка 8.* Функции find_last_file{) в качестве параметра передается расширение файла (либо `.txt`, либо `.answer`). Она находит в каталоге `$survey_dir` последний по алфавиту файл заданного типа. Эта функция общего назначения впоследствии используется функцией `get_file_contents()`, а также программой записи результатов опроса, которую мы рассмотрим в следующем разделе. Если в данном каталоге нет файлов такого типа, то функция `find_last_file()` возвращает значение `undef`.

- *Строка 20.* Функции get_file_contents(} в качестве параметра также передается расширение файла (опять-таки `.txt` либо `.answer`). Она возвращает содержимое последнего файла указанного типа из каталога опроса. Чтобы найти имя этого файла, она вызывает функцию `find_last_file()`.

Оставшаяся часть программы, приведенная в листинге 24.5, достаточно проста.

Листинг 24.5. Отображение опроса, часть II)

```
35:
36: # Получить содержимое последнего текстового файла с анкетой
37: my($question, @answers)=get_file_contents("txt");
38:
39: print header;
40: print qq{<FORM ACTION="/cgi/writesurvey.cgi" METHOD=POST>\n};
41: print "Вопрос: $question<P>\n";
42: my $answer=0;
43: foreach(@answers) {
44:     print "<INPUT TYPE=RADIO NAME=answer value=$answer>";
45:     print "$_<BR>\n";
46:     $answer++;
47: }
48: print qq{<INPUT TYPE=SUBMIT VALUE="Запрос">};
49: print qq{</FORM>};
```

ОСНОВНОЙ КОД программы начинается со строки 36. Содержимое последнего `.txt`-файла загружается в переменную `$question` для первой строки и в `@answers` — для остальной части файла функцией get_file_contents!).

В строке 40 вы должны изменить путь /cgi/writesurvey.cgi к CGI-программе записи данных опроса.

Далее выводится заголовок анкеты и ее вопросы посылаются броузеру. Каждая строка в массиве @answers оформляется в виде переключателя. Первой по порядку кнопке переключателя присваивается 0, второй — 1 и т.д. до тех пор, пока в массиве @answers не останется ни одного варианта ответа. Тело анкеты будет выглядеть примерно так:

```
<INPUT TYPE=RADIO NAME=answer value=0>Собака<BR>
<INPUT TYPE=RADIO NAME=answer value=1>Кошка<BR>
<INPUT TYPE=RADIO NAME=answer value=2>Рыбки<BR>
<INPUT TYPE=RADIO NAME=answer value=3> Они меня раздражают<BR>
```

После того как пользователь выберет вариант ответа и щелкнет на кнопке подачи запроса, CGI-программе будет передан параметр answer, содержащий номер ответа. CGI-программа /cgi/writesurvey.cgi, обрабатывающая данные формы (полный путь к ней задается в строке 40), запишет ответы в файл. Эта программа рассматривается в следующем разделе.

Часть II: анализ результатов

После того как пользователь щелкнет на кнопке подачи запроса (Submit), начинается настоящая работа. Ответ пользователя (т.е. выбранный им вариант) необходимо записать в файл, а результаты — свести в таблицу, а затем отобразить.

Программа, листинг которой приведен ниже, кажется довольно длинной, но на самом деле большую ее часть составляют подпрограммы, с которыми вы уже встречались. Подпрограммы блокирования файлов get_lock() и release_lock(), которые использовались на протяжении всей книги, и подпрограммы get_file_contents() и find_last_file() из программы отображения опроса также вносят существенный вклад в увеличение размера данной CGI-программы.

Начало кода программы обработки опроса приведено в листинге 24.6. Повторяю еще раз: пусть ее размер вас не пугает, так как большая часть этой программы вам уже знакома.

Листинг 24.6. Обработка опроса, часть I

```
#!/usr/bin/perl -w

use strict;
use Fcntl qw(:flock);
use CGI qw(:all);

my($survey_dir, $lockfile);
$survey_dir="/web/htdocs/poll";
$lockfile="/tmp/surveylock";

sub find_last_file {
    my($type)=@_;
    my(@files, $last_file);
        # Открыть каталог, взять последний файл
        # нужного типа.
        opendir(SD, $survey_dir) ||
           die "Ошибка при открытии $survey_dir: $!";
        @files=reverse sort grep(/\.$type$/, readdir SD);
        closedir(SD);
```

```
            $last_file=$files[$#files];
        return($last_file);
}

sub get_file_contents {
            my($type)=@_;
            my(@answers, $last_file);

        $last_file=find_last_file($type);

        return if (not defined $last_file);
            # Открыть файл и прочитать его содержимое
            open(QF, "$survey_dir/$last_file")
                  || die "Ошибка при чтении $last_file: $!";
            @answers=<QF>;
            close(QF);
            chomp @answers;  # Удалить символы новой строки
            return(@ answers);
}
sub get_lock {
    open(SEM, ">$lockfile")
        || die "Ошибка при создании семафора: $!";
    flock(SEM, LOCK_EX)
        || die "Ошибка при блокировании файла: $!";
}
sub release_lock {
    close(SEM);
}
```

Пока все в листинге 24.6 должно быть вам понятно и знакомо. Используемые здесь подпрограммы либо взяты из предыдущей программы, как, например, get_file_contents() и find_last_file(), либо это подпрограммы get_lock() и release_lock(). И опять-таки проверьте, Что Web-сервер может осуществлять запись в каталог, имя которого указанно в переменной $survey_dir).

И поскольку здесь все просто, давайте перейдем к продолжению программы, приведенному в листинге 24.7.

Листинг 24.7. Обработка опроса, часть II

```
my($question, @poss_answers)=get_file_contents("txt");

print header;

# Добавить данные в файл ответов
if (defined param("answer")) {
    my($lastfile);
    get_lock();
    # Найти имя последнего файла опроса и создать
    # на его основе имя файла ответов.
    $lastfile=find_last_file("txt");
    $lastfile=~s/txt/answer/;

    open(ANS, ">>$survey_dir/$lastfile")
        || die "Ошибка при открытии $lastfile: $!";
```

```
61:     print ANS param("answer"), "\n";
62:     close(ANS);
63:     release_lock();
64: }
65:
66: my(@answers)=get_file_contents("answer");
67: my(%results);
68: # Подсчитаем количество ответов с помощью хэша
70: foreach(@answers) {
71:     $results{$_}++;
72: }
73: my $ansno=0;
74: foreach my $ans (@poss_answers) {
75:     $results{$ansno}=0 if (! exists $results{$ansno});
76:     print "Ответ '$ans' был выбран $results{$ansno} раз<BR>";
77:     $ansno++;
78: }
```

Основная программа начинается со строки 46. В этой строке данные текущего опроса сохраняются в переменных `$question` и `@poss_answers`.

Начиная со строки 50 программа проверяет, дал ли пользователь ответы на вопросы. Не забывайте, что пользователь может просто щелкнуть на кнопке подачи запроса без выбора варианта ответа. Если ответ дан, то в строке 53 выполняется блокировка с помощью функции `get_lock()`, чтобы файл результатов мог в одно и то же время обновлять только один человек (а никак не несколько!).

В строке 56 находится последний в каталоге .txt-файл опроса — скажем, first.txt. Его расширение `.txt` меняется на `.answer`, т.е. в итоге получается имя файла `first.answer`. Далее этот файл открывается для записи, в него вносятся данные текущего опроса, а затем с помощью функции release_lock{)с файла снимается блокировка. Теперь в него могут спокойно вносить данные другие пользователи.

В строке 66 функция get_file_contents() используется для получения результатов опроса. Создается хэш с именем `%results`; его ключами являются номера ответов — 0, 1, 2 и т.д., а значениями — количество случаев появления каждого ответа.

Начиная со строки 74 выполняется вывод ответов. Если в хэше `%results` нет элемента, соответствующего данному конкретному ответу, то результату присваивается значение 0. Сам ответ и количество случаев его появления выводятся в строке 76.

И это все. На рис. 24.4 показаны данные результатов опроса. Если вы хотите улучшить их внешний вид, то можете сделать так, чтобы CGI-программа отображала ответы (и результаты) в виде разноцветных таблиц и со всеми другими характерными особенностями, благодаря которым HTML-страниицы выглядят так красиво и привлекательно.



> Вообще говоря, чтобы рассмотренная нами CGI-программа работала, каталог, где хранятся данные опроса (в нашем примере это `/web/htdocs/poll`), должен быть доступен для записи всем пользователям. В системе Windows NT можно определить свойства этого каталога так, чтобы он был доступен для записи только учетной записи типа guest. В системе UNIX с помощью команды `chmod` нужно будет установить для каталога значение прав доступа, равное 0777. Кроме того, если вы запускаете программу опроса из командной строки, то она может создать файл типа `.answer`, который окажется недоступным для записи Web-серверу. В подобном случае для организации нормальной работы программы составления отчета вам, возможно, придется удалить этот файл.

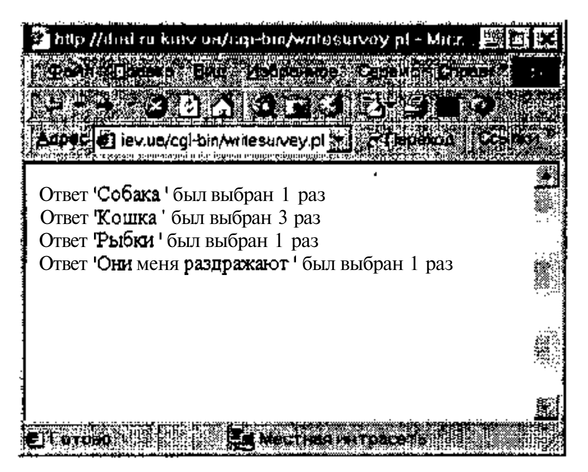


Рис. 24.4. Отображение данных результатов опроса

Резюме

На этом занятии мы рассмотрели ряд программ, способных придать вашим Web-страницам некоторое разнообразие. Мы изучили программу выборки содержимого других серверов и представления собранной информации на собственной Web-странице. Вы получили несколько полезных советов, касающихся заимствования информации, авторские права на которую принадлежат другим людям. И, наконец, мы рассмотрели программу проведения опроса, позволяющую вовлечь посетителей вашего Web-сервера в его работу.

Вопросы и ответы

Представляет ли угрозу для безопасности наличие на Web-сервере каталога, доступного для записи всем пользователям?

Да, хотя эта угроза невелика. Если ваш сервер был настроен разумно, то никто не сможет выгрузить информацию на ваш узел. Но, конечно, если ваш сервер позволяет кому угодно выгрузить что угодно и куда угодно, то возможность для злоупотребления есть. Если хотите, то для решения этой проблемы можете одновременно с .txt-файлами создать .answer-файлы. Только не забудьте с помощью функции `chmod` сделать сами .answer-файлы доступными для записи всем пользователям.

Правда ли, что против меня могут возбудить дело только за то, что я позаимствовал с другого сервера всего лишь заголовки?

Да, могут. Такие случаи уже были. В феврале 1999 года началось судебное разбирательство между фирмами Microsoft и Ticketmaster, возбужденное именно в такой ситуации. Microsoft якобы использовала "глубокие ссылки" на содержимое сервера Ticketmaster, и фирма Ticketmaster подала иск. В этом случае не было нарушения авторских прав, но наличие "глубоких ссылок" стало достаточным основанием для возбуждения дела. А в случае нарушения авторского права ситуация была бы гораздо более неприятной.

Я знаю один сервер, из которого хотел бы выбирать заголовки; он чем-то похож на **Slashdot. Но на нем нет XML-файла, который легко поддается** анализу. **Вместо него мне приходится использовать обычный HTML-файл. Как же мне провести анализ?**

Если вы собираетесь анализировать HTML-файл, *не* пытайтесь использовать для этого регулярные выражения, составленные самостоятельно. Анализировать HTML далеко не так просто, как кажется, и получить правильный результат почти невозможно. И даже если вы умудритесь с помощью регулярных выражений проанализировать часть HTML-кода, это сработает не во всех случаях. В CPAN есть модули для анализа HTML. Все они находятся в разделе HTML, т.е. имеют вид HTML::*.

Семинар

Контрольные вопросы

1. Что необходимо сделать для **извлечения** HTML-файла с Web-сервера?

   а) использовать модуль LWP,

   б) открыть сетевое соединение с сервером и "вытянуть" данные;

   в) использовать команду 'lynx -dump' или 'netscape -print'.

2. Если функция get модуля **LWP::Simple** завершилась неудачей, то что она вернет?

   а) сообщение об ошибке "No Document";

   б) пустую строку (т.е. "");

   в) значение **undef**.

Ответы

1. Правильный ответ — вариант а). Хотя варианты б) и в) тоже годятся, но они не так надежны и просты в использовании.
2. Правильный ответ — вариант в). См. объяснение, следующее за листингом 24.2.

Упражнения

- Результаты опроса вы можете отобразить в виде гистограммы, даже не используя графические модули. Для этого вам нужен .gif-файл размером 1x1 пиксель соответствующего цвета. Чтобы создать столбик, просто отобразите этот .gif-файл с соответствующими значениями высоты и ширины следующим образом:
  <IMG SRC="small.gif" **HEIGHT=20 WIDTH=200 alt="гистограмма"**

  Большинство фафических броузеров автоматически выполнят масштабирование этого маленького **.gif-файла** до нужного размера.

  Ваша задача: сделать так, чтобы программа опроса печатала его результаты в виде гистограммы. Чтобы определить длину столбца гистограммы, нужно

подсчитать количество всех голосов, а затем разделить на это число количество голосов по каждой категории. Например, если всего проголосовало 100 человек и за одну из категорий подано 40 голосов, то получим коэффициент 0,4, на который и нужно умножать длину соответствующего столбца.

- Результаты опроса, программа анализа данных которого приведена в листинге 24.4, легко могут быть искажены людьми, проголосовавшими более одного раза. Подумайте, как это можно предотвратить? Можно создать файл, в котором будут сохраняться адреса всех респондентов, проверять по этому файлу адрес очередного голосующего и не допускать дублирования. Однако это не позволит выполнить голосование людям, работающих через proxy-сервер (об этом шла речь на 23-м занятии, "Push-технология и счетчики посещений Web-страниц"). Поэтому придумайте метод, позволяющий человеку, посетившему ваш сервер, проголосовать только один раз.

  (Но только имейте в виду, что подобный метод не может быть простым. Считайте это, по большей части, чисто теоретическим экспериментом.)



Приложение

Инсталляция модулей в Perl

Устанавливать модули в Perl несложно и научиться это делать необходимо, если вы хотите овладеть искусством программирования на Perl. В этом приложении содержится информация о том, как устанавливать нужные вам модули.



Подробная информация о том, как устанавливать модули практически в любой операционной системе, содержится в документации по Perl. В документе, который называется `perlmodinstall`, содержатся даже инструкции по инсталляции модулей в таких непопулярных операционных системах, как OS/2 и VMS.

Выбор нужного модуля

Сначала вам необходимо выбрать нужный модуль. Хорошей отправной точкой для этого является библиотека CPAN, доступ к которой можно получить по адресу http://www.perl.com/CPAN. Вы должны решить, какой модуль вас интересует.

Имена модулям CPAN даны примерно в соответствии с их функциями. Например, модуль Image::Size берет изображение и сообщает, каков размер этого изображения; этот модуль используется для работы с Web-страницами. Но некоторые модули имеют необычные имена. Так, имя модуля `LWP` происходит от названия библиотеки Perl, которая называется `libwww-perl`.

Наборы модулей можно также найти в CPAN. В этих наборах содержится несколько связанных между собой модулей и обычно несколько сопутствующих модулей, причем все они находятся в одном большом пакете. Например, набор libnet устанавливается как модуль, но в процессе установки получается несколько модулей и все они предназначены для работы в сети. Например, модуль `LWP` — это часть набора libnet.



При установке модуля вы автоматически получаете всю документацию к нему.

Инсталляция модулей в системе...

В каждом из примеров, приведенных в следующих разделах, мы будем пытаться устанавливать модуль Date::Manip из библиотеки CPAN. Чтобы установить собственный модуль или набор модулей, просто подставьте имя этого модуля вместо `Date::Manip`.

...Windows 95/98/NT

Для системы Windows самым простым решением будет использование стандартных пакетов модулей, которые созданы ActiveState Tool Corp, конечно, при условии, что вы используете интерпретатор Perl этой фирмы.

Чтобы установить стандартный модуль в системе Windows, нужно сначала запустить Perl Package Manager (PPM). Эта утилита упрощает процесс инсталляции, так как она предоставляет для этой цели интерактивный интерфейс. Чтобы запустить PPM, нужно открыть окно сеанса MS-DOS; при этом вы должны быть подключены к Internet.

После командного приглашения просто наберите `ppm`, как показано ниже. В результате должна запуститься утилита PPM; если этого не произойдет, поищите файл `ppm.bat`, который был установлен вместе с версией ActiveState Perl и запустите его, указав полный путь.

```
C:\Windows>ppm
PPM interactive shell (1.1.3) - type 'help' for available commands.
PPM>
```

Чтобы найти конкретный модуль, воспользуйтесь командой search, как показано ниже. Эта команда нужна потому, что в ActiveState нет заранее построенных пакетов для всех модулей из CPAN; они есть только для наиболее популярных модулей. Кроме того, чтобы инсталлировать модуль, вы должны правильно указать его имя, не сделав при этом орфографических ошибок.

```
PPM> search Date
Packages available from http://www.ActiveState.com/packages
        Date-Calc
        Date-Manip
        TimeDate
PPM>
```

После того как вы найдете нужный модуль — в нашем примере это Date::Manip, — можете установить его с помощью команды install следующим образом:

```
PPM> install Date-Manip
Install package 'Date-Manip'?(y/N): y
Installing C:\Perl\html\lib\Date\Manip.html
Installing C:\Perl\htmlhelp\pkg-Date-Manip.html
Installing C:\Perl\htmlhelp\pkg-Date-Manip.hhc
Installing C:\Perl\site\lib\Date\Manip.pm
Writing C:\Perl\site\lib/auto/Date-Manip/.packlist
PPM>
```

Теперь модуль Date::Manip установлен!

Если вы хотите загрузить набор модулей и установить его вручную (например, если вы не имеете доступа к Internet или ваша локальная сеть защищена брандмауэром), то инструкции по загрузке и установке модулей вручную, а также о том, как обращаться с бранд-

мауэрами, можно найти на Web-сервере ActiveState по адресу http://vww.ActiveState.com. Фирма ActiveState поддерживает список часто задаваемых вопросов, касающийся распространения Perl, и здесь вы найдете все необходимые сведения и инструкции.



Инсталляция модулей без PPM, например с помощью компилятора С в системе Microsoft Windows, выходит за рамки данной книги. Поскольку Perl распространяется по методу открытого кода, то в его поставку входят инструкции о том, как самостоятельно построить Perl в системе Windows, но это работа не для новичков. Если же вы способны это осуществить, то самостоятельное построение модулей уже не окажется для вас слишком сложным, поскольку эти процедуры практически идентичны.

...UNIX, с помощью CPAN

Инсталляция модулей в системе UNIX может оказаться захватывающей, полной проблем или удивительно простой процедурой. Вам понадобится компилятор ANSI С (прекрасно подойдет тот, который использовался для построения Perl) и лицензия на его использование (если этого требует фирма-продавец). Вам понадобится также экземпляр программы архиватора GNU `gzip/gunzip`; некоторые фирмы, продающие систему UNIX, выпускают его в виде стандартной утилиты. Если у вас нет этой программы, можете получить ее экземпляр по адресу http://www.fsf.org.



Некоторые фирмы, продающие систему UNIX (в особенности, Hewlett-Packard), выпускают свои операционные системы вместе с компилятором С, который не является ANSI-совместимым. Это сильно урезанная версия настоящего компилятора, поэтому вам придется заплатить за настоящий компилятор или бесплатно загрузить и инсталлировать GNU компилятор С.

И еще одно: вам понадобятся права доступа пользователя root на том компьютере, где устанавливается Perl. Обычно Perl инсталлируется в качестве системной утилиты. Поэтому для доступа к его каталогам требуются соответствующие права.

В поставку Perl входит модуль под именем CPAN, который предназначен для помощи в установке других модулей. Чтобы начать установку, запустите с помощью Perl оболочку модуля CPAN следующим образом:

```
$ perl -MCPAN -e shell
```

Когда вы первый раз дадите эту команду, модуль CPAN спросит вас, откуда вы хотите "вытягивать" модули Perl и как хотите их устанавливать. В большинстве случаев достаточно выбирать стандартные ответы, принимаемые по умолчанию. Вас спросят о том, где находятся временные каталоги, какой зеркальный сервер CPAN вы хотите использовать (список будет представлен) и подключены ли вы к Internet через proxy-сервер.

Когда CPAN закончит задавать вам вопросы, появится следующее приглашение:

```
cpan shell — CPAN exploration and modules installation (v1.52)
ReadLine support available (try '"install Bundle::CPAN"}

cpan>
```

В ответ на это приглашение вы можете ввести команду i */шаблон/*, чтобы поискать информацию о пакете. Здесь параметр */шаблон/* определяет шаблон для поиска. Например, чтобы найти модуль Date: :Manip, введите следующую команду:

```
cpan> i /Manip/
```

Модулю CPAN может понадобиться связаться с сервером CPAN и получить новый экземпляр индекса. Это происходит только в случае необходимости, и весь процесс занимает некоторое время. После выполнения запроса CPAN выдает примерно следующую информацию:

```
Distribution    S/SB/SBECK/DateManip-5.39.tar.gz
Module          Date::Manip      (S/SB/SBECK/DateManip-5.39.tar.gz)
```

Чтобы инсталлировать модуль, наберите

```
cpan> install Date::Manip
```

В результате модуль CPAN выполнит для вас процесс загрузки, компиляции, тестирования и инсталляции модуля. На экране появится куча непонятной информации, которая выглядит примерно так, как показано ниже в очень сокращенном примере (комментариев # обычно нет, они добавлены здесь для пояснения).

```
Running make for S/SB/SBECK/DateManip-5.39.tar.gz
Fetching with Net::FTP:  # Загрузка модуля
  ftp://ftp.cpan.org/CPAN/authors/id/S/SB/SBECK/DateManip-5.39.tar.gz
Fetching with Net::FTP:  # Проверка контрольной суммы
  ftp://ftp.cpan.org/CPAN/authors/id/S/SB/SBECK/CHECKSUMS
Checksum for /root/.cpan/sources/authors/id/S/SB/SBECK/DateManip-5.39.tar.gz ok
DateManip-5.39/
DateManip-5.39/t/
DateManip-5.39/t/date_date_0.t
DateManip-5.39/t/Manip.cnf
DateManip-5.39/t/date.t

  CPAN.pm: Going to build S/SB/SBECK/DateManip-5.39.tar.gz

Checking if your kit is complete...
Looks good
Writing Makefile for Date::Manip
mkdir blib       # Построение модуля
mkdir blib/lib
mkdir blib/lib/Date
cp Manip.pod blib/lib/Date/Manip.pod
cp Manip.pm blib/lib/Date/Manip.pm
mkdir blib/arch
mkdir blib/arch/auto
mkdir blib/arch/auto/Date
mkdir blib/arch/auto/Date/Manip
mkdir blib/lib/auto/Date
mkdir blib/lib/auto/Date/Manip
mkdir blib/man3
Manifying blib/man3/Date::Manip.3
  /usr/bin/make  — OK
Running make test  # Тестирование модуля
PERL_DL_NONLAZY=1 /usr/bin/perl -Iblib/arch -Iblib/lib -I/usr/local/lib/perl5/5.
6.0/i386-freebsd -I/usr/local/lib/perl5/5.6.0 -e 'use Test::Harness qw(&runtests
 $verbose); $verbose=0; runtests @ARGV;' t/*.t
t/date..............ok
t/date_date_0.......ok
Files=31, Tests=839, 153 wallclock secs (139.67 cusr + 5.69 csys = 145.36 CPU)
```

```
  /usr/bin/make test — OK
Running make install  i Установка модуля в системе
Installing /usr/local/lib/perl5/site_perl/5.6.0/Date/Manip.pod
Installing /usr/local/lib/perl5/site_perl/5.6.0/Date/Manip.pm
Installing /usr/local/man/man3/Date::Manip.3
Writing /usr/local/lib/perl5/site_perl/5.6.0/i386-freebsd/auto/Date/Manip/.packl
ist
Appending installation info to /usr/local/lib/perl5/5.6.0/i386-freebsd/perllocal
.pod
  /usr/bin/make install  ~ OK
```

Но, конечно, полученная вами информация будет существенно отличаться от приведенной выше. Итак, теперь модуль протестирован и установлен. Кричите "ура"!

...UNIX, трудным способом

Хотя в системе UNIX можно установить модули, не пользуясь модулем CPAN, в большинстве случаев не нужно поступать подобным образом. Данный метод приведен здесь только для полноты изложения. А на самом деле следует использовать модуль CPAN везде, где это только возможно.

Для начала вы должны вручную загрузить модуль из библиотеки CPAN. Он представляет собой архив tar, упакованный с помощью программы gzip. Например, если этот модуль — Date: :Calc, то нужно получить его самую последнюю версию, которая называется примерно так: Date-Calc-X.T.tar.gz. После того как вы загрузили данный архив, зайдите в каталог, в котором он находится, и распакуйте его следующим образом:

```
$ gunzip Date-Calc-4.3.tar.gz
$ tar xf Date-Calc-4.3.tar.gz
```

При распаковке создается подкаталог Date-Calc-4.3. Перейдите в него с помощью команды cd и введите следующую команду:

```
$ perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Date::Calc
```

В результате будет создан makefile, который необходим для процесса построения модуля. После этого постройте модуль, воспользовавшись командой make следующим образом:

```
$ make
mkdir blib
mkdir blib/lib
...
Manifying blib/man3/Date::Calc.3
  /usr/bin/make  — OK
```

Этот процесс может занять некоторое время.

На следующем этапе нужно протестировать модуль, чтобы узнать, правильно ли он построен. Для этого просто наберите команду make test следующим образом:

```
$ make test
PERL_DL_NONLAZY=1 /usr/bin/perl -Iblib/arch -Iblib/lib -I/usr/local/lib/perl5/5.
6.0/i386-freebsd -I/usr/local/lib/perl5/5.6.0 -e 'use Test::Harness qw(&runtests
```

```
$verbose); $verbose=0; runtests @ARGV;' t/*.t
t/f000..............ok
t/f001..............ok
...
t/f033..............ok
All tests successful.
Files=34, Tests=1836, 11 wallclock secs ( 7.65 cusr +  1.10 csys =  8.75 CPU)
  /usr/bin/make test ~ OK
```

Для того чтобы удостовериться, что модуль построен корректно, *всегда* следует использовать команду make test. Это позволит сэкономить вам (и другим) многие часы отладки в дальнейшем. После завершения тестирования нужно инсталлировать модуль, как показано ниже. Этот шаг обычно выполняется с помощью учетной записи root, так как при инсталляции производится запись в системные каталоги.

```
$ su
Password: *******
i make install
Installing /usr/local/lib/perl5/site_perl/5.6.0/i386-freebsd/auto/Date/Calc/Calc
.so
...
Appending installation info to /usr/local/lib/perl5/5.6.0/i386-freebsd/perllocal
.pod
  /usr/bin/make install  — OK
```

Теперь все готово!

Инсталляция модулей на компьютере Macintosh

Инсталляция модулей на компьютере Macintosh — это трудная задача. Просмотрите FAQ-файл сервера MacPerl, чтобы получить информацию о дроплетах (droplet), которые можно использовать для инсталляции модулей. FAQ-файл сервера MacPerl находится по адресу http://www.macperl.com.

Что делать, если вам не разрешается устанавливать модули

ЕСЛИ ВЫ можете устанавливать программы на компьютере, то можете устанавливать и модули. Ваша способность это сделать зависит от того, насколько сложным является модуль и согласны ли вы немного помучиться с его установкой. Иногда системный администратор не разрешает устанавливать модуль, потому что не хочет, чтобы его использовали другие. В некоторых случаях особые модули нужны только вам или небольшой группе людей, поэтому устанавливать их так, чтобы они были доступны всем пользователям, просто нецелесообразно.

В любом случае установить собственные экземпляры модулей Perl в собственных каталогах совсем несложно.

Во-первых, вам нужно построить модуль с помощью приведенной выше инструкции, но с небольшим исключением. Вы можете указать программе установки, что нужно поместить модули в особый каталог. Если вы используете программу PPM для

системы Microsoft Windows, то, прежде чем инсталлировать модуль, вы должны указать этой программе, что хотите выполнить установку в другой каталог. Это делается с помошью команды set следующим образом:

```
PPM> set root c:\myperl
PPM> set build c:\myperl
```

В результате модули будут транслироваться и инсталлироваться в каталог c:\myperl.

В системе UNIX, когда вы используете модуль CPAN, можете указать каталог установки с помощью параметра makepl_arg следующим образом:

```
cpan> o conf makepl_arg PREFIX="/home/clintp/perl/lib"
```

Или, если вы устанавливаете модули вручную с помошью утилиты make, можете указать каталог установки, использовав во время построения аргумент PREFIX в первой команде.

```
$ perl Makefile.PL PREFIX="/home/clintp/perl/lib"
```

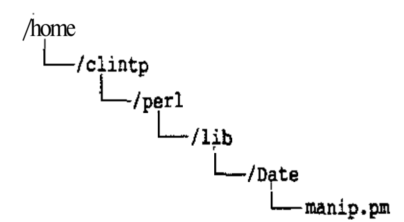
Каким бы методом вы ни воспользовались, модуль, который вы пытаетесь инсталлировать, будет помешен в каталог /home/clintp/perl/lib/. Затем, если нужно, можно переместить этот модуль в другой каталог.



Будьте внимательны и не перемещайте модули с одного компьютера на другой, если эти компьютеры принадлежат к разным типам. Скомпилированный модуль, как и сам Perl, будет работать только на компьютерах одного типа. Кроме того, постарайтесь не перемещать модули между различными версиями Perl; иногда это приводит к тому, что они перестают работать. В подобной ситуации вам придется переустановить модуль.

Использование модулей, установленных в необычных местах

Чтобы использовать модули, установленные в каталоге, отличном от стандартного, нужно применить директиву use lib. Например, если вы устанавливаете модуль Date::Manip в каталог /home/clintp/perl/lib/ с помощью инструкций из предыдущего раздела, то у вас получится древовидная файловая структура, аналогичная той, которая показана ниже на рисунке.



Чтобы воспользоваться модулем, в начале программы нужно ввести следующий код:

```
use lib '/home/clintp/perl/lib'; # Подключаем нестандартные модули
else
use Date::Manip;
```

Тогда Perl, прежде чем проводить поиск по своим каталогам, будет искать модуль в указанном каталоге. Этот метод можно использовать также при инсталляции на своем компьютере более новых версий модулей для тестирования, не затирая старые версии и не создавая проблемы с несовместимостью.

Предметный указатель

B

C

D

H

I

M

P

R

S

T

U

W

X

А

Б

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Я

Учебное пособие

Клинтон Пирс

Освой самостоятельно Perl за 24 часа

Издательский дом "**Вильямс**".
101509, Москва, ул. Лесная, д. 43, стр. 1.
Изд. лиц. ЛР № 090230 от 23.06.99
Госкомитета РФ по печати.

Подписано в печать 12.02.01. Формат 70x100/16.
Гарнитура Times. Бумага газетная. Печать офсетная.
Уч.-изд. л. 21,86. Усл. печ. л. 26,23. Доп. тираж 4000 экз. **Заказ** № 15.

Отпечатано с фотоформ в ФГУП «Печатный двор» Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.